Tsvyatko Konov

Nikolay Nedyalkov

Nikolay Vassilev

Teodor Stoev

Nikolay Kostov

Hristo Germanov

Yosif Yosifov

Radoslav Todorov

Pavlina Hadjieva

Teodor Bozhikov

Svetlin Nakov,

Veselin Kolev

& Co.

Radoslav Ivanov

Yordan Pavlov

Iliyan Murdanliev

Radoslav Kirliov

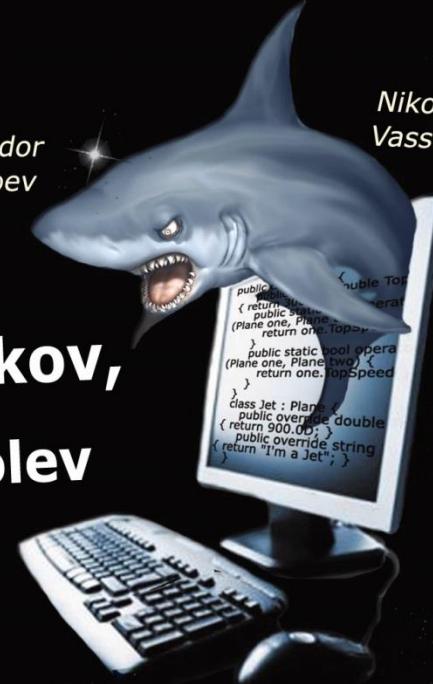
Mira Bivas

Stefan Staev

Vesselin Georgiev

Pavel Donchev

Mihail Stoynov

Mihail Valkov

Dilyan Dimitrov

Stanislav Zlatinov

Fundamentals of Computer Programming

The Bulgarian C# Book

Contents

Contents 2
Detailed Table of Contents 5
Preface 13
Chapter 1. Introduction to Programming 69
Chapter 2. Primitive Types and Variables 111
Chapter 3. Operators and Expressions 139
Chapter 4. Console Input and Output 165
Chapter 5. Conditional Statements 195
Chapter 6. Loops 211
Chapter 7. Arrays 235
Chapter 8. Numeral Systems 265
Chapter 9. Methods 293
Chapter 10. Recursion 351
Chapter 11. Creating and Using Objects 385
Chapter 12. Exception Handling 415
Chapter 13. Strings and Text Processing 457
Chapter 14. Defining Classes 499
Chapter 15. Text Files 615
Chapter 16. Linear Data Structures 641
Chapter 17. Trees and Graphs 681
Chapter 18. Dictionaries, Hash-Tables and Sets 727
Chapter 19. Data Structures and Algorithm Complexity 769
Chapter 20. Object-Oriented Programming Principles 807
Chapter 21. High-Quality Programming Code 853
Chapter 22. Lambda Expressions and LINQ 915
Chapter 23. Methodology of Problem Solving 935
Chapter 24. Sample Programming Exam – Topic #1 985
Chapter 25. Sample Programming Exam – Topic #2 1041
Chapter 26. Sample Programming Exam – Topic #3 1071
Conclusion 1119

FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#

(The Bulgarian C# Programming Book)

Svetlin Nakov & Co.

Dilyan Dimitrov
Hristo Germanov
Iliyan Murdanliev
Mihail Stoynov
Mihail Valkov
Mira Bivas
Nikolay Kostov
Nikolay Nedyalkov
Nikolay Vasilev
Pavel Donchev
Pavlina Hadjieva
Radoslav Ivanov
Radoslav Kirilov
Radoslav Todorov
Stanislav Zlatinov
Stefan Staev
Svetlin Nakov
Teodor Bozhikov
Teodor Stoev
Tsvyatko Konov
Vesselin Georgiev
Veselin Kolev
Yordan Pavlov
Yosif Yosifov

Sofia, 2013

FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#

(The Bulgarian C# Programming Book)

Official Web Site:

http://www.introprogramming.info

ISBN 978-954-400-773-7

Detailed Table of Contents

Contents 2
Detailed Table of Contents 5
Preface 13
About the Book 13
C# and .NET Framework 17
How To Read This Book? 22
Why Are Data Structures and Algorithms Emphasized? 25
Do You Really Want to Become a Programmer? 26
A Look at the Book's Contents 29
History: How Did This Book Come to Be? 38
Authors and Contributors 40
The Book Is Free of Charge! 53
Reviews 53
License 63
Resources Coming with the Book 65
Chapter 1. Introduction to Programming 69
In This Chapter 69
What Does It Mean "To Program"? 69
Stages in Software Development 71
Our First C# Program 75
The C# Language and the .NET Platform 79
Visual Studio IDE 93
Alternatives to Visual Studio 104
Decompiling Code 104
C# in Linux, iOS and Android 107
Other .NET Languages 107
Exercises 108
Solutions and Guidelines 108
Chapter 2. Primitive Types and Variables 111
In This Chapter 111
What Is a Variable? 111
Data Types 111
Variables 123
Value and Reference Types 128
Literals 131

Exercises....135
Solutions and Guidelines....136
Chapter 3. Operators and Expressions....139
In This Chapter....139
Operators....139
Type Conversion and Casting....152
Expressions....158
Exercises....160
Solutions and Guidelines....161
Chapter 4. Console Input and Output....165
In This Chapter....165
What Is the Console?....165
Standard Input-Output....169
Printing to the Console....169
Console Input....183
Console Input and Output – Examples....190
Exercises....192
Solutions and Guidelines....193
Chapter 5. Conditional Statements....195
In This Chapter....195
Comparison Operators and Boolean Expressions....195
Conditional Statements "if" and "if-else"....200
Conditional Statement "switch-case"....206
Exercises....208
Solutions and Guidelines....209
Chapter 6. Loops....211
In This Chapter....211
What Is a "Loop"?....211
While Loops....211
Do-While Loops....216
For Loops....221
Foreach Loops....225
Nested Loops....226
Exercises....231
Solutions and Guidelines....233
Chapter 7. Arrays....235
In This Chapter....235
What Is an "Array"?....235
Declaration and Allocation of Memory for Arrays....235
Access to the Elements of an Array....238
Reading an Array from the Console....241

Printing an Array to the Console 243
Iteration through Elements of an Array 244
Multidimensional Arrays 246
Arrays of Arrays 253
Exercises....................257
Solutions and Guidelines 259
Chapter 8. Numeral Systems 265
In This Chapter 265
History in a Nutshell 265
Numeral Systems 266
Representation of Numbers 276
Exercises....................289
Solutions and Guidelines 290
Chapter 9. Methods 293
In This Chapter 293
Subroutines in Programming....................293
What Is a "Method"?....................293
Why to Use Methods? 294
How to Declare, Implement and Invoke a Method? 295
Declaring Our Own Method 295
Implementation (Creation) of Own Method 300
Invoking a Method....................301
Parameters in Methods 303
Returning a Result from a Method 328
Best Practices when Using Methods 345
Exercises....................347
Solutions and Guidelines 348
Chapter 10. Recursion 351
In This Chapter 351
What Is Recursion?....................351
Example of Recursion 351
Direct and Indirect Recursion....................352
Bottom of Recursion 352
Creating Recursive Methods....................352
Recursive Calculation of Factorial 353
Recursion or Iteration?....................355
Simulation of N Nested Loops 356
Which is Better: Recursion or Iteration? 362
Using Recursion – Conclusions 378
Exercises....................378
Solutions and Guidelines 380
Chapter 11. Creating and Using Objects 385

In This Chapter 385
Classes and Objects 385
Classes in C# 387
Creating and Using Objects 390
Namespaces 405
Exercises 410
Solutions and Guidelines 412
Chapter 12. Exception Handling 415
In This Chapter 415
What Is an Exception? 415
Exceptions Hierarchy 424
Throwing and Catching Exceptions 426
The try-finally Construct 432
IDisposable and the "using" Statement 437
Advantages of Using Exceptions 439
Best Practices when Using Exceptions 445
Exercises 453
Solutions and Guidelines 454
Chapter 13. Strings and Text Processing 457
In This Chapter 457
Strings 457
Strings Operations 462
Constructing Strings: the StringBuilder Class 480
String Formatting 488
Exercises 491
Solutions and Guidelines 496
Chapter 14. Defining Classes 499
In This Chapter 499
Custom Classes 499
Usage of Class and Objects 502
Organizing Classes in Files and Namespaces 505
Modifiers and Access Levels (Visibility) 508
Declaring Classes 509
The Reserved Word "this" 511
Fields 512
Methods 518
Accessing Non-Static Data of the Class 519
Hiding Fields with Local Variables 522
Visibility of Fields and Methods 524
Constructors 531
Properties 549
Static Classes and Static Members 559

Structures 580
Enumerations 584
Inner Classes (Nested Classes) 590
Generics 594
Exercises..610
Solutions and Guidelines 613
Chapter 15. Text Files...615
In This Chapter 615
Streams 615
Reading from a Text File 620
Writing to a Text File 628
Input / Output Exception Handling 630
Text Files – More Examples 631
Exercises..636
Solutions and Guidelines 638
Chapter 16. Linear Data Structures 641
In This Chapter 641
Abstract Data Structures 641
List Data Structures...642
Exercises..676
Solutions and Guidelines 678
Chapter 17. Trees and Graphs 681
In This Chapter 681
Tree Data Structures 681
Trees 681
Graphs...714
Exercises..722
Solutions and Guidelines 723
Chapter 18. Dictionaries, Hash-Tables and Sets 727
In This Chapter 727
Dictionary Data Structure 727
Hash-Tables 735
The "Set" Data Structure...760
Exercises..765
Solutions and Guidelines 767
Chapter 19. Data Structures and Algorithm Complexity 769
In This Chapter 769
Why Are Data Structures So Important?...769
Algorithm Complexity 770
Comparison between Basic Data Structures 779
When to Use a Particular Data Structure? 779

Choosing a Data Structure – Examples 786
External Libraries with .NET Collections 801
Exercises 803
Solutions and Guidelines 804
Chapter 20. Object-Oriented Programming Principles 807
In This Chapter 807
Let's Review: Classes and Objects 807
Object-Oriented Programming (OOP) 807
Fundamental Principles of OOP 808
Inheritance 809
Abstraction 824
Encapsulation 828
Polymorphism 830
Cohesion and Coupling 836
Object-Oriented Modeling (OOM) 842
UML Notation 844
Design Patterns 847
Exercises 851
Solutions and Guidelines 852
Chapter 21. High-Quality Programming Code 853
In This Chapter 853
Why Is Code Quality Important? 853
What Does Quality Programming Code Mean? 854
Why Should We Write Quality Code? 854
Identifier Naming 857
Code Formatting 866
High-Quality Classes 874
High-Quality Methods 878
Proper Use of Variables 883
Proper Use of Expressions 890
Use of Constants 891
Proper Use of Control Flow Statements 894
Defensive Programming 898
Code Documentation 900
Code Refactoring 904
Unit Testing 905
Additional Resources 912
Exercises 912
Solutions and Guidelines 913
Chapter 22. Lambda Expressions and LINQ 915
In This Chapter 915
Extension Methods 915

Anonymous Types 918
Lambda Expressions 920
LINQ Queries 924
Nested LINQ Queries 930
LINQ Performance 930
Exercises....933
Solutions and Guidelines 933
Chapter 23. Methodology of Problem Solving 935
In This Chapter 935
Basic Principles of Solving Computer Programming Problems 935
Use Pen and Paper 936
Generate Ideas and Give Them a Try! 937
Decompose the Task into Smaller Subtasks 938
Verify Your Ideas! 941
If a Problem Occurs, Invent a New Idea! 943
Choose Appropriate Data Structures! 946
Think about the Efficiency! 950
Implement Your Algorithm! 953
Write the Code Step by Step! 954
Test Your Solution! 967
General Conclusions 979
Exercises....980
Solutions and Guidelines 983
Chapter 24. Sample Programming Exam – Topic #1 985
In This Chapter 985
Problem 1: Extract Text from HTML Document 985
Problem 2: Escape from Labyrinth 1012
Problem 3: Store for Car Parts 1026
Exercises....1038
Solutions and Guidelines 1040
Chapter 25. Sample Programming Exam – Topic #2 1041
In This Chapter 1041
Problem 1: Counting the Uppercase / Lowercase Words in a Text 1041
Problem 2: A Matrix of Prime Numbers 1054
Problem 3: Evaluate an Arithmetic Expression 1060
Exercises....1069
Solutions and Guidelines 1069
Chapter 26. Sample Programming Exam – Topic #3 1071
In This Chapter 1071
Problem 1: Spiral Matrix 1071
Problem 2: Counting Words in a Text File 1078
Problem 3: School 1099

Preface

If you want to take up **programming** seriously, you've come across **the right book**. For real! This is the book with which you can make your first steps in programming. It will give a flying start to your long journey into learning modern programming languages and software development technologies. This book teaches the **fundamental principles and concepts of programming**, which have not changed significantly in the past 15 years.

Do not hesitate to read this book even if C# is not the language you would like to pursue. Whatever language you move on to, the knowledge we will give you here will stick, because this book will teach you to think like programmers. We will show you and teach you **how to write programs for solving practical algorithmic problems**, form the skills in you to come up with (and implement) algorithms, and use various data structures.

As improbable as it might seem to you, the basic principles of writing computer programs have not changed all that much in the past 15 years. Programming languages change, technologies get modernized, integrated development environments get more and more advanced but **the fundamental principles of programming remain the same**. When beginners learn to think algorithmically, and then learn to divide a problem instinctively into a series of steps to solve it, as well as when they learn to select the appropriate data structures and write high-quality programming code that is when they become programmers. Once you acquire these skills, you can easily learn new languages and various technologies – like Web programming, HTML5 and JavaScript, mobile development, databases and SQL, XML, REST, ASP.NET, Java EE, Python, Ruby and hundreds more.

About the Book

This book is designed specifically to teach you to think like a programmer and the C# language is just a tool that can be replaced by any other modern programming languages, such as Java, C++, PHP or Python. **This is a book on programming, not a book on C#!**

Please Excuse Us for the Bugs in the Translation!

This book was originally **written in Bulgarian** language by a large team of volunteer software engineers and later **translated into English**. None of the authors, translators, editors and the other contributors is a native English speaker so you might find many mistakes and imprecise translation. **Please, excuse us!** Over 70 people have participated in this project (mostly Bulgarians): authors, editors, translators, correctors, bug submitters, etc. and

still the quality could be improved. The entire team congratulates you on your choice to read this book and we believe the content in it is more important that the small mistakes and inaccuracies you might find. Enjoy!

Who Is This Book Aimed At?

This book is **best suited for beginners**. It is intended for anyone who so far has not engaged seriously in programming and would like to begin doing it. This book starts from scratch and introduces you step by step into the fundamentals of programming. It won't teach you absolutely everything you might need for becoming a software engineer and working at a software company, but it will lay the groundwork on which you can build up technological knowledge and skills, and through them you will be able to turn programming into your profession.

If you've never written a computer program, don't worry. There is always a first time. In this book we will **teach you how to program from scratch**. We do not expect any previous knowledge or abilities. All you need is some basic computer literacy and a desire to take up programming. The rest you will learn from the book.

If you can already write simple programs or if you have studied programming at school or in college, or you've coded with friends, **do not assume you know everything**! Read this book and you'll become aware of how many things you've missed. This book is indeed for beginners, but it **teaches concepts** and skills that even experienced professional programmers lack. Software companies are riddled with a shocking amount of self-taught amateurs who, despite having programmed on a salary for years, have no grasp of the fundamentals of programming and have no idea what a hash table is, how polymorphism works and how to work with bitwise operations. Don't be like them! Learn the **basics of programming first** and then the technologies. Otherwise you risk having your programming skills crippled, more or less, for years, if not for life.

If, on the other hand, you have programming experience, examine this book in details and see if you are familiar with all subjects we have covered, in order to decide whether it is for you or not. Take a close look especially at the chapters "Data Structures and Algorithms Complexity", "Object-Oriented Programming Principles", "Methodology of Problem Solving" and "High-Quality Programming Code". It is very likely that, even if you have several years of experience, you might not be able to work well with **data structures**; you might not be able to evaluate the **complexity of an algorithm**; you might not have mastered in depth the concepts of **object-oriented programming** (including UML and design patterns); and you might not be acquainted with the best practices for writing **high-quality programming code**. These are very important topics that are not covered in all books on programming, so don't skip them!

Previous Knowledge Is Not Required!

In this book **we do not expect any previous programming knowledge** from the readers. It is not necessary for you to have studied information technology or computer science, in order to read and comprehend the book content. The book **starts from scratch** and gradually gets you involved in programming. All technical terms you will come across will have been explained beforehand and it is not necessary for you to know them from other sources. If you don't know what a compiler, debugger, integrated development environment, variable, array, loop, console, string, data structure, algorithm, algorithm complexity, class or object are, don't be alarmed. From this book, you will learn all these terms and many more and gradually get accustomed to using them constantly in your everyday work. **Just read the book consistently and do the exercises.**

Certainly, if, after all, you do have prior knowledge in computer science and information technologies, they will by all means be of use to you. If, at university, you major in the field of computer science or if you study information technology at school, this will only help you, but it is not a must. If you major in tourism, law or other discipline that has little in common with computer technology, **you could still become a good programmer**, as long as you have the desire. The software industry is full of good developers without a computer science or related degree.

It is expected for you **to have basic computer literacy**, since we would not be explaining what a file, hard disk and network adapter is, nor how to move the mouse or how to write on a keyboard. We expect you to know how to work with a computer and how to use the Internet.

It is recommended that the readers have at least some basic knowledge of **English**. The entire documentation you will be using every day and almost all of the websites on programming you would be reading at all times are in English. In the profession of a programmer, **English is absolutely essential**. The sooner you learn it, the better. We hope that you already speak English; otherwise how do you read this text?

	Make no illusion you can become a programmer without learning even a little English! This is simply a naive expectation. If you don't speak English, complete a course of some sort and then start reading technical literature, make note of any unfamiliar words and learn them. You will see for yourselves that Technical English is easy to learn and it doesn't take much time.

What Is the Scope of This Book?

This book covers the **fundamentals of programming**. It will teach you how to define and use variables, how to work with primitive data structures (such as numbers), how to organize logical statements, conditional statements and

loops, how to print on the console, how to use arrays, how to work with numeral systems, how to define and use methods, and how to create and use objects. Along with the **basic programming knowledge**, this book will help you understand more **complicated concepts** such as string processing, exception handling, using complex data structures (like trees and hash tables), working with text files, defining custom classes and working with LINQ queries. The concepts of object-oriented programming (OOP) – an established approach in modern software development – will be covered in depth. Finally, you'll be faced with the practices for **writing high-quality programs** and solving real-world programming problems. This book presents a complete methodology for solving programming problems, as well as algorithmic problems in general, and shows how to implement it with a few sample subjects and programming exams. This is something you will not find in any other book on programming!

What Will This Book Not Teach You?

This book will not award you **the profession "software engineer"**! This book won't teach you how to use the entire .NET platform, how to work with databases, how to create dynamic web sites and develop mobile applications, how to create window-based graphical user interface (GUI) and rich Internet applications (RIA). You won't learn how to develop complex software applications and systems like Skype, Firefox, MS Word or social networks like Facebook and retail sites like Amazon.com. And no other single book will. These kinds of projects require **many, many years of work** and experience and the knowledge in this book is just a wonderful beginning for the future programmer geek.

From this book, you won't learn software engineering, team work and you won't be able to prepare for working on real projects in a software company. In order to learn all of this, you will need a few more books and extra courses, but do not regret the time you will spend on this book. You are making the right choice by **starting with the fundamentals of programming** rather than directly with Web development, mobile applications and databases. This gives you the opportunity to **become a master programmer** who has in-depth knowledge of programming and technology. After you acquire the fundamentals of programming, it will become much easier for you to read and learn databases and web applications, and you will understand what you read much easier and in greater depth rather than if you directly begin learning SQL, ASP.NET, AJAX, XAML or WinRT.

Some of your colleagues directly begin programming with Web or mobile applications and databases without knowing what an array, a list or hash table is. Do not envy them! They have set out to do it the hard way, backwards. They will learn to make low-quality websites with PHP and MySQL, but they will find it **infinitely difficult to become real professionals**. You, too, will learn web technologies and databases, but before you take them up, **learn how to program!** This is much more important. Learning one

technology or another is very easy once you know the basics, when you can think algorithmically and you know how to tackle programming problems.

Starting to program with web applications or/and databases is just as incorrect as studying up a foreign language from some classical novel rather than from the alphabet and a textbook for beginners. It is not impossible, but if you lack the basics, it is much more difficult. It is highly-probable that you would end up lacking vital fundamental knowledge and being the laughing-stock of your colleagues/peers.

How Is the Information Presented?

Despite the large number of authors, co-authors and editors, we have done our best to make the style of the book similar in all chapters and highly comprehensible. The content is presented in a **well-structured** manner; it is broken up into many titles and subtitles, which make its reception easy and looking up information in the text quick.

The present book is **written by programmers for programmers**. The authors are active software developers, colleagues with genuine experience in both software development and training future programmers. Due to this, the quality of the content presentation is at a very good level, as you will see for yourself.

All authors are distinctly aware that the **sample source code** is one of the most important things in a book on programming. Due to this very reason, the text is accompanied with many, many examples, illustrations and figures.

When every chapter is written by a different author, there is no way to completely avoid differences in the style of speech and the quality of chapters. Some authors put a lot of work (for months) and a lot of efforts to **make their chapters perfect**. Others could not invest too much effort and that is why some chapters are not as good as the best ones. Last but not least, the **experience** of the authors varies – some have been programming professionally for 2-3 years, while others – for 15 years. This affects the quality, no doubt, but we assure you that **every chapter has been reviewed** and meets the quality standards of Svetlin Nakov and his team.

C# and .NET Framework

This book is about **programming**. It is intended to teach you to think as a programmer, to write code, to think in data structures and algorithms and to solve problems.

We use **C#** and **Microsoft .NET Framework** (the platform behind C#) only as means for writing programming code and we do not scrutinize the language's specifics. This same book can be found in versions for other languages like Java and C++, but the differences are not very significant.

Nevertheless, let's give a short account of C# (pronounced "see sharp").

	C# is a modern programming language for development of software applications.

If the words "C#" and ".NET Framework" are unknown to you, you'll learn in details about them and their connection in the next chapter. Now let's explain briefly what C#, .NET, .NET Framework, CLR and the other technologies related to C# are.

The C# Programming Language

C# is a **modern object-oriented, general-purpose programming language**, created and developed by Microsoft together with the .NET platform. There is highly diverse software developed with C# and on the .NET platform: office applications, web applications, websites, desktop applications, mobile applications, games and many others.

C# is a **high-level language** that is similar to Java and C++ and, to some extent, languages like Delphi, VB.NET and C. All C# programs are object-oriented. They consist of a set of definitions in classes that contain methods and the methods contain the program logic – the instructions which the computer executes. You will find out more details on what a class, a method and C# programs are in the next chapter.

Nowadays C# is **one of the most popular programming languages**. It is used by millions of developers worldwide. Because C# is developed by Microsoft as part of their modern platform for development and execution of applications, the .NET Framework, the language is widely spread among Microsoft-oriented companies, organizations and individual developers. For better or for worse, as of this book writing, the **C# language** and the **.NET platform** are maintained and **managed entirely by Microsoft** and are not open to third parties. Because of this, all other large software corporations like IBM, Oracle and SAP base their solutions on the Java platform and use Java as their primary language for developing their own software products.

Unlike C# and the .NET Framework, the **Java language and platform are open-source** projects that an entire community of software companies, organizations and individual developers take part in. The standards, the specifications and all the new features in the world of Java are developed by workgroups formed out of the entire Java community, rather than a single company (as the case of C# and .NET Framework).

The C# language is distributed together with a special environment on which it is executed, called the **Common Language Runtime (CLR)**. This environment is part of the platform .NET Framework, which includes CLR, a bundle of standard libraries providing basic functionality, compilers, debuggers and other development tools. Thanks to the framework CLR programs are portable and, once written they can function with little or no changes on various hardware platforms and operating systems. C# programs

are most commonly run on MS Windows, but the .NET Framework and CLR also support mobile phones and other portable devices based on Windows Mobile, Windows Phone and Windows 8. C# programs can still be run under Linux, FreeBSD, iOS, Android, MacOS X and other operating systems through the free .NET Framework implementation **Mono**, which, however, is not officially supported by Microsoft.

The Microsoft .NET Framework

The C# language is not distributed as a standalone product – it is a part of the Microsoft .NET Framework platform (pronounced "Microsoft dot net framework"). **.NET Framework** generally consists of an environment for the development and execution of programs, written in C# or some other language, compatible with .NET (like VB.NET, Managed C++, J# or F#). It consists of:

- the .NET programming **languages** (C#, VB.NET and others);
- an environment for the execution of managed code (**CLR**), which executes C# programs in a controlled manner;
- a set of **development tools**, such as the **csc** compiler, which turns C# programs into intermediate code (called MSIL) that the CLR can understand;
- a set of **standard libraries**, like **ADO.NET**, which allow access to databases (such as MS SQL Server or MySQL) and **WCF** which connects applications through standard communication frameworks and protocols like HTTP, REST, JSON, SOAP and TCP sockets.

The .NET Framework is part of every modern Windows distribution and is available in different versions. The latest version can be downloaded and installed from Microsoft’s website. As of this book’s publishing, the latest version of the **.NET Framework is 4.5**. Windows Vista includes out-of-the-box .NET Framework 2.0, Windows 7 – .NET 3.5 and Windows 8 – .NET 4.5.

Why C#?

There are many reasons why we chose C# for our book. It is a modern programming language, widely spread, **used by millions of programmers** around the entire world. At the same time C# is a very simple and **easy to learn** (unlike C and C++). It is natural to start with a language that is suitable for beginners while still widely used in the industry by many large companies, making it one of the **most popular programming languages** nowadays.

C# or Java?

Although this can be extensively discussed, it is commonly acknowledged that **Java is the most serious competitor to C#**. We will not make a comparison between Java and C#, because C# is undisputedly the better,

more powerful, richer and just better engineered. But, for the purposes of this book, we have to emphasize that any modern programming language will be sufficient to learn programming and algorithms. We chose C#, because it is **easier to learn** and is distributed with highly convenient, free integrated development environment (e.g. Visual C# Express Edition). Those who prefer Java can prefer to use the Java version of this book, which can be found here: www.introprogramming.info.

Why Not PHP?

With regards to programing languages popularity, besides C# and Java, another widely used language is **PHP**. It is suitable for developing small web sites and web applications, but it gives rise to serious difficulties when implementing large and complicated software systems. In the software industry PHP is used first and foremost **for small projects**, because it can easily lead developers into writing code that is bad, disorganized and hard to maintain, making it inconvenient for more substantial projects. This subject is also debatable, but it is commonly accepted that, because of its antiquated concepts and origins it is built on and because of various evolutionary reasons, **PHP is a language that tends towards low-quality programming**, writing bad code and creating hard to maintain software. PHP is a procedural language in concept and although it supports the paradigms of modern object-oriented programming, most PHP programmers write procedurally. PHP is known as the language of "code monkeys" in the software engineering profession, because most PHP programmers write **terrifyingly low-quality code**. Because of the tendency to write low-quality, badly structured and badly organized programming code, the entire concept of the PHP language and platform is considered wrong and serious companies (like Microsoft, Google, SAP, Oracle and their partners) avoid it. Due to this reason, if you want to become a serious software engineer, start with C# or Java and **avoid PHP** (as much as possible).

Certainly, **PHP has its uses in the world of programming** (for example creating a blog with WordPress, a small web site with Joomla or Drupal, or a discussion board with PhpBB), but the entire PHP platform is **not well-organized** and engineered for large systems like .NET and Java. When it comes to non-web-based applications and large industrial projects, PHP is not by a long shot among the available options. Lots and lots of experience is necessary to use PHP correctly and to develop high-quality professional projects with it. PHP developers usually learn from tutorials, articles and low-quality books and pick up bad practices and habits, which then are hard to eradicate. Therefore, **do not learn PHP as your first development language**. **Start with C# or Java**.

Based on the large experience of the authors' collective we advise you to begin programming with C# and ignore languages such as C, C++ and PHP until the moment you have to use them.

Why Not C or C++?

Although this is also debatable, the **C and C++ languages are considered rather primitive, old and decaying**. They still have their uses and are suitable for low-level programming (e.g. programming for specialized hardware devices), but we do not advise you to pursue them.

You can program in pure C, if you have to write an operating system, a hardware device driver or if you want to program an embedded device, because of the lack of alternatives and the need to control the hardware very carefully. **The C language is morally old** and in no way do we advise you to begin programming with it. A programmer's productivity under pure C is many times lower compared to their productivity under modern general-purpose programming languages like C# and Java. A variant of C is used among Apple / iPhone developers, but not because it is a good language, but because there is no decent alternative. Most Apple-oriented programmers do not like Objective-C, but they have no choice in writing in something else.

C++ is acceptable when you have to program applications that require very **close work with the hardware** or that have special **performance requirements** (like 3D games). For all other purposes (like web applications development or business software) C++ is phenomenally inadequate. We do not advise you to pursue it, if you are starting with programming just now. The reason it is still being studied in some schools and universities is hereditary, because these institutions are very conservative. For example, the International Olympiad in Informatics (IOI) continues to promote C++ as the only language permitted to use at programming contests, although **C++ is rarely used in the industry**. If you don't believe this, look through some job search site and count the percentage of job advertisements with C++.

The C++ language lost its popularity mainly because of the inability to quickly write quality software with it. In order to write high-quality software in C++, you have to be an incredibly smart and experienced programmer, whereas the same is not strictly required for C# and Java. **Learning C++ takes much more time** and very few programmers know it really well. The productivity of C++ programmers is many times lower than C#'s and that is why C++ is losing ground. Because of all these reasons, **the C++ language is slowly fading away** and therefore we do not advise you to learn it.

Advantages of C#

C# is an **object-oriented** programming language. Such are all modern programming languages used for serious software systems (like Java and C++). The advantages of object-oriented programming are brought up in many passages throughout the book, but, for the moment, you can think of object-oriented languages as languages that allow working with objects from the real world (for example student, school, textbook, book and others). Objects have properties (e.g. name, color, etc.) and can perform actions (e.g. move, speak, etc.).

By starting to program with C# and the .NET Framework platform, you are on a **very perspective track**. If you open a website with job offers for programmers, you'll see for yourself that the demand for C# and .NET specialists is huge and is close to the demand for Java programmers. At the same time, the demand for PHP, C++ and other technology specialists is far lower than the demand for C# and Java engineers.

For the good programmer, the language they use is of no significant meaning, because they know **how to program**. Whatever language and technology they might need, they will master it quickly. Our goal is **not to teach you C#, but rather teach you programming!** After you master the fundamentals of programming and learn to think algorithmically, when you acquaint with other programming languages, you will see for yourself how much in common they have with C# and how easy it will be to learn them. Programming is built upon principles that change very slowly over the years and this book teaches you these very principles.

Examples Are Given in C# 5 and Visual Studio 2012

All examples in this book are with regard to **version 5.0** of the C# language and the **.NET Framework 4.5** platform, which is the latest as of this book's publishing. All examples on using the Visual Studio integrated development environment are with regard to version **2012** of the product, which were also the latest at the time of writing this book.

The Microsoft **Visual Studio 2012** integrated development environment (IDE) has a free version, suitable for beginner C# programmers, called Microsoft **Visual Studio Express 2012** for Windows Desktop. The difference between the free and the full version of Visual Studio (which is a commercial software product) lies in the availability of some functionalities, which we will not need in this book.

Although we use C# 5 and Visual Studio 2012, most examples in this book will work flawlessly under .NET Framework 2.0 / 3.5 / 4.0 and C# 2.0 / 3.5 / 4.0 and **can be compiled under Visual Studio 2005 / 2008 / 2010**.

It is of no great significance which version of C# and Visual Studio you'll use while you learn programming. What matters is that you learn **the principles of programming and algorithmic thinking**! The C# language, the .NET Framework platform and the Visual Studio integrated development environment are just tools and you can exchange them for others at any time. If you read this book and VS2012 is not currently the latest, be sure almost all of this book's content will still be the same due to backward compatibility.

How To Read This Book?

Reading this book has to be accompanied with **lots and lots of practice**. You won't learn programming, if you don't practice! It would be like trying to learn how to swim from a book without actually trying it. There is no other way!

The more you work on the problems after every chapter, the more you will learn from the book.

Everything you read here, you would have to try for yourself on a computer. Otherwise you won't learn anything. For example, once you read about Visual Studio and how to write your first simple program, you must by all means download and install Microsoft Visual Studio (or Visual C# Express) and try to write a program. Otherwise you won't learn! In theory, everything seems easy, but **programming means practice**. Remember this and try to solve the problems from this book. They are carefully selected – they are neither too hard to discourage you, nor too easy, so you'll be motivated to perceive solving them as a challenge. If you encounter difficulties, look for help at the **discussion group** for the "C# Programming Fundamentals" training course at Telerik Software Academy: http://forums.academy.telerik.com (the forum is intended for Bulgarian developers but the people "living" in it speak English and will answer your questions regarding this book, don't worry). Thousands students solve the exercises from this book every year so you will find many solutions to each problem from the book. We will also publish official solutions + tests for every exercise in the book at its web site.

	Reading this book without practicing is meaningless! You must spend much more time on writing programs than reading the text itself. It is just like learning to drive: no one can learn driving by reading books. To learn driving, you need to drive many times in different situations, roads, cars, etc. To learn programming, you need to program!

Everybody has studied math in school and knows that learning how to solve math problems requires lots of practice. No matter how much they watch and listen to their teachers, **without actually sitting down and solving problems, they won't learn**. The same goes for programming. You need lots of practice. You need to write a lot, to solve problems, to experiment, to endeavor in and to struggle with problems, to make mistakes and correct them, to try and fail, to try anew and experience the moments when things finally work out. You need lots and lots of practice. This is the only way you will make progress.

So people say that to become a developer you might need to write at least 50,000 – 100,000 lines of code, but the correct number can vary a lot. Some people are fast learners or just have problem-solving experience. Others may need more practice, but in all cases **practicing programming is very important**! You need to solve problems and to write code to become a developer. There is no other way!

Do Not Skip the Exercises!

At the end of each chapter there is a considerable list of **exercises**. **Do not skip them!** Without exercises, you will not learn a thing. After you read a chapter, you should sit in front of the computer and **play with the examples**

you have seen in the book. Then you should set about solving all problems. If you cannot solve them all, you should at least try. If you don't have all the time necessary, you must at least attempt solving the first few problems from each chapter. Do not carry on without **solving problems after every chapter**, it would just be meaningless! The problems are small feasible situations where you apply the stuff you have read. In practice, once you have become programmers, you would solve similar problems every day, but on a larger and more complex scale.

	You must at all cost strive to solve the exercise problems after every chapter from the book! Otherwise you risk not learning anything and simply wasting your time.

How Much Time Will We Need for This Book?

Mastering the fundamentals of programming is a crucial task and **takes a lot of time**. Even if you're incredibly good at it, there is no way that you will learn programming on a good level for a week or two. To learn any human skill, you need to read, see or be shown how it is done and then try doing it yourselves and practice a lot. The same goes for programming – you must either read, see or listen how it is done, then **try doing it yourself**. Then you would succeed or you would not and you would try again, until you finally realize you have learned it. Learning is done step by step, consecutively, in series, with a lot of effort and consistency.

If you want to read, understand, learn and acquire thoroughly and in-depth the subject matter in this book, you have to invest **at least 2 months for daylong activity** or at least 4-5 months, if you read and exercise a little every day. This is the minimum amount of time it would take you to be able to grasp in depth the fundamentals of programming.

The necessity of such an amount of lessons is confirmed by the free trainings at Telerik Software Academy (http://academy.telerik.com), which follow this very book. The hundreds of students, who have participated in trainings based on the lectures from this book, usually learn all subjects from this book within **3-4 months of full-time work**. Thousands of students every year solve all exercise problems from this book and successfully sit on programming exams covering the book's content. Statistics shows that anyone without prior exposure to programming, who has spent less than the equivalent of 3-4 months daylong activity on this book and the corresponding courses at Telerik Academy, fails the exams.

The main subject matter in the book is presented in more than **1100 pages**, which will take you a month (daylong) just to read them carefully and test the sample programs. Of course, you have to spend enough time on the exercises (few more months); without them you would hardly learn programming.

Exercises: Complex or Easy?

The exercises in the book consist of about **350 problems** with varying difficulty. For some of them you will need a few minutes, for others several hours (if you can solve them at all without help). This means you would need a month or two of daylong exercising or several months, if you do it little by little.

The exercises at each chapter are ordered in **increasing level of difficulty**. The first few exercises are easy, similar to the examples in the chapter. The last few exercises are usually complex. You might need to use external resources (like information from Wikipedia) to solve them. Intentionally, the last few exercises in each chapter require **skills outside of the chapter**. We want to push you to perform a search in your favorite search engine. You need to **learn searching on the Internet!** This is an essential skill for any programmer. You need to learn how to learn. Programming is about learning every day. Technologies constantly change and you can't know everything. To be a programmer means to **learn new APIs, frameworks, technologies and tools every day**. This cannot be avoided, just prepare yourself. You will find many problems in the exercises, which require searching on the Internet. Sometimes you will need the skills from the next chapter, sometimes some well-known algorithm, sometimes something else, but in all cases **searching on the Internet is an essential skill** you need to acquire.

Solving the exercises in the book takes a few **months**, really. If you don't have that much time at your disposal, ask yourselves if you really want to pursue programming. This is a very serious initiative in which you must invest a really great deal of efforts. If you really want to learn programming on a good level, **schedule enough time** and follow the book or the video lectures based on it.

Why Are Data Structures and Algorithms Emphasized?

This book teaches you, in addition to the basic knowledge in programming, proper **algorithmic thinking** and using basic **data structures** in programming. Data structures and algorithms are a programmer's most important fundamental skills! If you have a good grasp of them, you will not have any trouble becoming proficient in any software technology, development tool, framework or API. That is what the most serious software companies rely on when hiring employees. Proof of this are job interviews at large companies like Google and Microsoft that rely exclusively on **algorithmic thinking** and knowledge of all basic **data structures and algorithms**.

The information below comes from **Svetlin Nakov**, the leading author of this book, who passed software engineering interviews at Microsoft and Google in 2007-2008 and shares his own experience.

Job Interviews at Google

100% of the questions at job interviews for software engineers at Google, Zurich, are about **data structures, algorithms and algorithmic thinking**. At such an interview you may have to implement on a white board a linked list (see the chapter "Linear Data Structures") or come up with an algorithm for filling a raster polygon (given in the form of a GIF image) with some sort of color (see Breadth-first search in the chapter "Trees and Graphs"). It seems like Google are interested in hiring people **who can think algorithmically** and who have a grasp of basic data structures and computer algorithms. Any technology that candidates would afterwards use in their line of work can be quickly learned. Needless to say, do not assume this book will give you all the knowledge and skills to pass a job interview at Google. The knowledge in the book is absolutely a necessary minimum, but not completely sufficient. It only marks the first steps.

Job Interviews at Microsoft

A lot of questions at job interviews for software engineers at Microsoft, Dublin, focus on **data structures**, **algorithms** and **algorithmic thinking**. For example, you could be asked to reverse the words in a string (see the chapter "Strings and Text Processing" or to implement topological sorting in an undirected graph (see the chapter "Trees and Graphs"). Unlike Google, Microsoft asks a lot of engineering questions related to software architectures, multithreading, writing secure code, working with large amounts of data and software testing. This book is far from sufficient for applying at Microsoft, but the knowledge in it will surely be of use to you for the majority of questions.

About the LINQ Technology

The book includes a chapter on the popular .NET technology **LINQ** (**Language Integrated Query**), which allows execution of various queries (such as searching, sorting, summation and other group operations) on arrays, lists and other objects. It is placed towards the end on purpose, after the chapters on **data structures** and **algorithms complexity**. The reason behind this is that the good programmer must know what happens when they sort a list or search in an array according to criteria and how many operations these actions take. If LINQ is used, it is not obvious how a given query works and how much time it takes. **LINQ is a very powerful and widely-used technology**, but it has to be mastered at a later stage (at the end of the book), after you are well familiar with the basics of programming, the main algorithms and data structures. Otherwise you risk learning how to write inefficient code without realizing how it works and how many operations it performs in the background.

Do You Really Want to Become a Programmer?

If you **want to become a programmer**, you have to be aware that true programmers are serious, persevering, thinking and questioning people who

handle all kinds of problems. It is important for them to master quickly all modern or legacy platforms, technologies, libraries, APIs, programming tools, programming languages and development tools necessary for their job and to feel programming as a part of their life.

Good programmers **spend an extraordinary amount of time on advancing their engineering skills**, on learning new technologies, new programming languages and paradigms, new ways to do their job, new platforms and new development tools every day. They are capable of **logical thinking**; reasoning on problems and coming up with algorithms for solving them; **imagining** solutions as a series of steps; **modeling** the surrounding world using technological means; **implementing** their ideas as programs or program components; **testing** their algorithms and programs; seeing issues; foreseeing the exceptional circumstances that can come about and handling them properly; listening to the advice of more experienced people; adapting their applications' user interface to the user's needs; adapting their algorithms to the capabilities of the machines and the environment they will be executed on and interacted with.

Good programmers **constantly read books, articles or blogs on programming** and are interested in new technologies; they constantly enrich their knowledge and constantly improve the way they work and the quality of software they write. Some of them become obsessed to such an extent that they even forget to eat or sleep when confronted with a serious problem or simply inspired by some interesting lecture or presentation. If you have the tendency to get **motivated to such an extent** to do something (like playing video games incessantly), you can learn programming very quickly by getting into the mindset that programming is the most interesting thing in this world for you, in this period of your life.

Good programmers have one or more computers, an Internet connection and **live in constant reach with technologies**. They regularly visit websites and blogs related to new technologies, communicate everyday with their colleagues, visit technology lectures, seminars and other events, even if they have no use for them at the moment. They experiment with or research the new means and new ways for making a piece of software or a part of their work. They examine new libraries, learn new languages, try new frameworks and play with new development tools. That way they **develop their skills** and maintain their level of awareness, competence and professionalism.

True programmers know that they can never master their profession to its full extent, because it constantly changes. They live with the firm belief **that they have to learn their entire lives**; they enjoy this and it satisfies them. True programmers are curious and questioning people that want to know how everything works – from a simple analog clock to a GPS system, Internet technology, programming languages, operation systems, compilers, computer graphics, games, hardware, artificial intelligence and everything else related to computers and technologies. The more they learn, the more knowledge and skills they crave after. **Their life is tied to technologies** and they change

with them, enjoying the development of computer science, technologies and the software industry.

Everything we tell you about true programmers, we know firsthand. We are convinced that **programmer is a profession that requires your full devotion** and complete attention, in order to be a really good specialist – experienced, competent, informed, thinking, reasoning, knowing, capable and able to deal with non-standard situations. Anyone who takes up programming "among other things" is fated to being a mediocre programmer. Programming requires **complete devotion for years**. If you are ready for all of this, continue reading and take into account that the next few months you will spend on this book on programming are just a small start. And then you will learn for years until you turn programming into your profession. Once that happens, you would still learn something every day and compete with technologies, so that you can maintain your level, until one day programming develops your thinking and skills enough, so that **you may take up another profession**, because few programmers reach retirement; but there are quite a lot of successful people who have begun their careers with programming.

Motivate Yourself to Become a Programmer or Find Another Job!

If you still haven't given up on **becoming a good programmer** and if you have already come to the understanding deep down that the next months and years will be tied every day to constant diligent work on mastering the secrets of programming, software development, computer science and software technologies, you may use an old technique for **self-motivation** and confident achievement of goals that can be found in many books and ancient teachings under one form or another. Keep imagining that you are programmers and that you have succeeded in becoming ones; **you engage every day in programming**; it is your profession; you can write all the software in the world (provided you have enough time); you can solve any problem that experienced programmers can solve. Keep thinking constantly and incessantly of your goal. Keep telling yourself, sometimes even out loud: "**I want to become a good programmer** and I have to work hard for this, I have to read a lot and learn a lot, I have to solve a lot of problems, every day, constantly and diligently". Put programming books everywhere around you, even stick a sign that says "**I'll become a good programmer**" by your bed, so that you can see it every evening when you go to bed and every morning when you wake up. **Program every day** (no exceptions!), solve problems, have fun, learn new technologies, experiment; try writing a game, making a website, writing a compiler, a database and hundreds of other programs you may come up with original ideas for. In order to become good programmers, **program every day** and think about programming every day and keep imagining the future moment when you are an excellent programmer. You can, as long as you deeply believe that you can! Everybody can, as long as they believe that they can and pursue their goals constantly

without giving up. No-one would motivate you better than yourselves. Everything depends on you and this book is your first step.

A great way to really learn programming is to program every day for a year. If you program every day (without exception) and you do it for a long time (e.g. year or two) there is no way to not become a programmer. Anyone who practices programming every day for years will become good someday. This is valid for any other skill: if you want to learn it, just practice every day for a long time.

A Look at the Book's Contents

Now let's take a glance at what we are about to encounter in the **next chapters of the book**. We will give an account of each of them with a few sentences, so that you know what you are about to learn.

Chapter 0: Preface

The preface (the current chapter) introduces the readers to the book, its content, **what the reader will learn** and what will not, how to read the book, why we use the C# language, why we focus on data structures and algorithms, etc. The preface also describes the history of the book, the content of its chapter one by one, the team of authors, editors and translators from Bulgarian to English. In contains the full reviews written by famous software engineers from Microsoft, Google, SAP, VMware, Telerik and other leading software companies from all over the world.

Author of the preface is **Svetlin Nakov** (with little contribution from Veselin Kolev and Mihail Stoynov). Translation to English: by Ivan Nenchovski (edited by Mihail Stoynov, Veselina Raykova, Yoan Krumov and Hristo Radkov).

Chapter 1: Introduction to Programming

In the chapter "Introduction to Programming", we will take a look at the basic terminology in programming and **write our first program**. We will familiarize ourselves with what programming is and what connection to computers and programming languages it has. We will briefly review the main stages in software development, introduce the **C# language**, the .NET platform and the different Microsoft technologies used in software development. We will examine what auxiliary tools we need to program in C# and use the C# language to write our **first computer program**, **compile** it and **run** it using the command line, as well as Microsoft **Visual Studio** integrated development environment. We will familiarize ourselves with the MSDN Library – the documentation for the .NET Framework, which will help us in our study of the language's capabilities.

Author of the chapter is **Pavel Donchev**; editors are Teodor Bozhikov and Svetlin Nakov. The content of the chapter is somewhat based on the work of

Luchesar Cekov from the book "Introduction to Programming with Java". Translation to English: by Atanas Valchev (edited by Vladimir Tsenev and Hristo Radkov).

Chapter 2: Primitive Types and Variables

In the chapter "Primitive Types and Variables", we will examine **primitive types and variables in C#** – what they are and how to work with them. First, we will focus on **data types** – integer types, real floating-point types, Boolean, character types, strings and object types. We will continue with **variables**, what they and their characteristics are, how to declare them, how they are assigned a value and what variable initialization is. We will familiarize ourselves with the main categories of data types in C# – value and reference types. Finally, we will focus on **literals**, what they are and what kinds of literals there are.

Authors of the chapter are **Veselin Georgiev** and **Svetlin Nakov**; editor is Nikolay Vasilev. The content of the entire chapter is based on the work of Hristo Todorov and Svetlin Nakov from the book "Introduction to Programming with Java". Translation to English: by Lora Borisova (edited by Angel Angelov and Hristo Radkov).

Chapter 3: Operators and Expressions

In the chapter "Operators and Expressions", we will familiarize ourselves with the **operators in C#** and the operations they perform on the various data types. We will clarify the priorities of operators and familiarize ourselves with the types of operators, according to the count of the arguments they take and the operations they perform. Then, we will examine **typecasting**, why it is necessary and how to work with it. Finally, we will describe and illustrate **expressions** and how they are utilized.

Authors of the chapter are **Dilyan Dimitrov** and **Svetlin Nakov**; editor is Marin Georgiev. The content of the entire chapter is based on the work of Lachezar Bozhkov from the book "Introduction to Programming with Java". Translation to English: by Angel Angelov (edited by Martin Yankov and Hristo Radkov).

Chapter 4: Console Input and Output

In the chapter "Console Input and Output", we will get familiar with the **console** as a means for **data input and output**. We will explain what it is, when and how it is used, what the concepts of most programming languages for accessing the console are. We will familiarize ourselves with some of the features in C# for user interaction and will examine the main streams for input-output operations **`Console.In`**, **`Console.Out`** and **`Console.Error`**, the class **`Console`** and the utilization of **format strings** for printing data in various formats. We will see how to convert text into a number (**parsing**), since this is the way to enter numbers in C#.

Author of the chapter is **Iliyan Murdanliev** and editor is Svetlin Nakov. The content of the entire chapter is largely based on the work of Boris Valkov from the book "Introduction to Programming with Java". Translation to English: by Lora Borisova (edited by Dyanko Petkov).

Chapter 5: Conditional Statements

In the chapter "Conditional Statements" we will cover the **conditional statements in C#**, which we can use to execute different actions depending on some condition. We will explain the syntax of the **conditional operators**: **`if`** and **`if-else`** with suitable examples and explain the practical applications of the selection control operator **`switch`**. We will focus on the best practices that must be followed, in order to achieve a better style of programming when utilizing nested or other types of conditional statements.

Author of the chapter is **Svetlin Nakov** and editor is Marin Georgiev. The content of the entire chapter is based on the work of Marin Georgiev from the book "Introduction to Programming with Java". Translation to English: by George Vaklinov (edited by Momchil Rogelov).

Chapter 6: Loops

In the chapter "Loops", we will examine the **loop mechanisms**, through which we can execute a snippet of code repeatedly. We will discuss how conditional repetitions (**`while`** and **`do-while`** loops) are implemented and how to work with **`for`** loops. We will give examples of the various means for defining a loop, the way they are constructed and some of their key applications. Finally, we will see how we can use multiple loops within each other (nested loops).

Author of the chapter is **Stanislav Zlatinov** and editor is Svetlin Nakov. The content of the entire chapter is based on the work of Rumyana Topalska from the book "Introduction to Programming with Java". Translation to English: by Angel Angelov (edited by Lora Borisova).

Chapter 7: Arrays

In the chapter "Arrays", we will familiarize ourselves with arrays as a means for **working with a sequence of elements** of the same type. We will explain what they are, how we can declare, create and instantiate arrays and how to provide access to their elements. We will examine **one-dimensional and multidimensional arrays**. We will learn the various ways for iterating through an array, reading from the standard input and writing to the standard output. We will give many exercises as examples, which can be solved using arrays, and show you how useful they are.

Author of the chapter is **Hristo Germanov** and editor is Radoslav Todorov. The content of the chapter is based on the work of Mariyan Nenchev from the book "Introduction to Programming with Java". Translation to English: by Boyan Dimitrov (edited by Radoslav Todorov and Zhelyazko Dimitrov).

Chapter 8: Numeral Systems

In the chapter "Numeral Systems", we will take a look at the means for working with various **numeral systems and the representation of numbers** in them. We will pay special attention to the way numbers are represented in **decimal**, **binary** and **hexadecimal** numeral systems, because they are widely used in computers, communications and programming. We will also explain the methods for encoding numeral data in a computer and the types of encodings, namely signed magnitude, one's complement, two's complement and binary-coded decimals.

Author of the chapter is **Teodor Bozhikov** and editor is Mihail Stoynov. The content of the entire chapter is based on the work of Petar Velev and Svetlin Nakov from the book "Introduction to Programming with Java". Translation to English: by Atanas Valchev (edited by Veselina Raykova).

Chapter 9: Methods

In the chapter "Methods", we will get to know in details the **subroutines in programming**, which are called **methods** in C#. We will explain when and why methods are used; will show how methods are declared and what a method signature is. We will learn how to **create** a custom method and how to use (**invoke**) it subsequently, and will demonstrate how we can use parameters in methods and how to return a **result** from a method. Finally, we will discuss some established practices when working with methods. All of this will be backed up with examples explained in details and with extra exercises.

Author of the chapter is **Yordan Pavlov**; editors are Radoslav Todorov and Nikolay Vasilev. The content of the entire chapter is based on the work of Nikolay Vasilev from the book "Introduction to Programming with Java". Translation to English: by Ivaylo Dyankov (edited by Vladimir Amiorkov and Franz Fischbach).

Chapter 10: Recursion

In the chapter "Recursion", we will familiarize ourselves with **recursion and its applications**. Recursion is a powerful programming technique where a **method invokes itself**. By means of recursion we can solve complicated **combinatorial problems** where we can easily exhaust different combinatorial configurations. We will demonstrate many examples of correct and incorrect recursion usage and we will convince you how useful it can be.

Author of the chapter is **Radoslav Ivanov** and editor is Svetlin Nakov. The content of the entire chapter is based on the work of Radoslav Ivanov and Svetlin Nakov from the book "Introduction to Programming with Java". Translation to English: by Vasya Stankova (edited by Yoan Krumov).

Chapter 11: Creating and Using Objects

In the chapter "Creating and Using Objects", we will get to know the basic concepts of object-oriented programming – **classes and objects** – and we will explain **how to use classes** from the standard libraries of the .NET Framework. We will focus on some commonly used system classes and will show how to create and use their instances (**objects**). We will discuss how to access **properties** of an object, how to call **constructors** and how to work with static fields in classes. Finally, we will focus on the term "namespaces" – how they help us, how to include and use them.

Author of the chapter is **Teodor Stoev** and editor is Stefan Staev. The content of the entire chapter is based on the work of Teodor Stoev and Stefan Staev from the book "Introduction to Programming with Java". Translation to English: by Vasya Stankova (edited by Todor Mitev).

Chapter 12: Exception Handling

In the chapter "Exception Handling", we will get to know **exceptions** in object-oriented programming and in C# in particular. We will learn how to **catch exceptions** using the **try-catch** clause, how to pass them to the calling methods and **how to throw** standard, custom or caught exceptions using the **throw** statement. We will give a number of examples of their utilization and will look at the types of exceptions and the **exceptions hierarchy** they form in the .NET Framework. Finally, we will look at the advantages of using exceptions and how to apply them in specific situations.

Author of the chapter is **Mihail Stoynov** and editor is Radoslav Kirilov. The content of the entire chapter is based on the work of Luchesar Cekov, Mihail Stoynov and Svetlin Nakov from the book "Introduction to Programming with Java". Translation to English: by Dimitar Bonev and George Todorov (edited by Doroteya Agayna).

Chapter 13: Strings and Text Processing

In the chapter "Strings and Text Processing", we will familiarize ourselves with **strings**: how they are implemented in C# and how we can process text content. We will go through different methods for **manipulating text**; and learn how to extract **substrings** according to passed parameters, how to **search** for keywords as well as how to **split a string** by separator characters. We will provide useful information on **regular expressions** and we will learn how to extract data matching a specific pattern. Finally, we will take a look at the methods and classes for achieving more elegant and strict **formatting** of text content on the console, with various ways for printing numbers and dates.

Author of the chapter is **Veselin Georgiev** and editor is Radoslav Todorov. The content of the entire chapter is based on the work of Mario Peshev from the book "Introduction to Programming with Java". Translation to English: by Vesselin Georgiev (edited by Todor Mitev and Vladimir Amiorkov).

Chapter 14: Defining Classes

In the chapter "Defining Classes", we will show how we can **define custom classes** and what the elements of a class are. We will learn to **declare fields**, **constructors** and **properties** in classes and will again recall what a method is but will broaden our knowledge on methods and their access modifiers. We will focus on the characteristics of **constructors** and we will explain in details how program objects exist in the heap (dynamic memory) and how their fields are initialized. Finally, will explain what class **static** elements – fields (including **constants**), **properties** and **methods** – are and how to utilize them. In this chapter, we will also introduce generic types (**generics**), enumerated types (**enumerations**) and nested classes.

Authors of the chapter are **Nikolay Vasilev**, **Svetlin Nakov**, **Mira Bivas** and **Pavlina Hadjieva**. The content of the entire chapter is based on the work of Nikolay Vasilev from the book "Introduction to Programming with Java". Translation to English: by Radoslav Todorov, Yoan Krumov, Teodor Rusev and Stanislav Vladimirov (edited by Vladimir Amiorkov, Pavel Benov and Nencho Nenchev). This is the largest chapter in the book, so lots of contributors worked on it to prepare it to a high quality standard for you.

Chapter 15: Text Files

In the chapter "Text Files", we will familiarize ourselves with **working with text files** in the .NET Framework. We will explain what a **stream** is, what its purpose is and how it is used. We will describe what a **text file** is and how to **read** and **write data** in text files and will present and elaborate on the best practices for catching and handling **exceptions** when working with text files. Naturally, we will visualize and demonstrate in practice all of this with a lot of examples.

Author of the chapter is **Radoslav Kirilov** and editor is Stanislav Zlatinov. The content of the entire chapter is based on the work of Danail Alexiev from the book "Introduction to Programming with Java". Translation to English: by Nikolay Angelov (edited by Martin Gebov).

Chapter 16: Linear Data Structures

In the chapter "Linear Data Structures", we will familiarize ourselves with some of the basic representations of data in programming and with **linear data structures**, because very often, in order to solve a given problem, we need to work with a **sequence of elements**. For example, to read this book we have to read consecutively every single page, e.g. we have to traverse consecutively every single element of its set of pages. We are going to see how for a specific problem some data structure is more efficient and convenient than another. Then we will examine the linear structures "**list**", "**stack**" and "**queue**" and their applications and will get to know in details some implementations of these structures.

Author of the chapter is **Tsvyatko Konov** and editors are Dilyan Dimitrov and Svetlin Nakov. The content of the entire chapter is largely based on the work of Tsvyatko Konov and Svetlin Nakov from the book "Introduction to Programming with Java". Translation to English: by Vasya Stankova (edited by Ivaylo Gergov).

Chapter 17: Trees and Graphs

In the chapter "Trees and Graphs", we will look at the so called **tree-like data structures**, which are **trees and graphs**. Knowing the properties of these structures is important for modern programming. Every one of these structures is used for modeling real-life problems that can be efficiently solved with their help. We will examine in details what tree-like data structures are and show their primary advantages and disadvantages. Also, we will provide sample implementations and exercises, demonstrating their practical utilization. Further, we will scrutinize binary trees, binary search trees and **balanced trees** and then examine the **data structure "graph"**, the types of graphs and their usage. We will also show which parts of the .NET Framework make use of **binary search trees**.

Author of the chapter is **Veselin Kolev** and editors are Iliyan Murdanliev and Svetlin Nakov. The content of the entire chapter is based on the work of Veselin Kolev from the book "Introduction to Programming with Java". Translation to English: by Kristian Dimitrov and Todor Mitev (edited by Nedjaty Mehmed and Dyanko Petkov).

Chapter 18: Dictionaries, Hash Tables and Sets

In the chapter "Dictionaries, Hash Tables and Sets", we will analyze more complex data structures like **dictionaries and sets**, and their implementations with **hash tables** and **balanced trees**. We will explain in details what hashing and hash tables mean, and why they are such important parts of programming. We will discuss the concept of "**collisions**" and how they can occur when implementing hash tables. We will also suggest various approaches for solving them. We will look at the abstract data structure "**set**" and explain how it can be implemented with a **dictionary** or a **balanced tree**. We will provide examples that illustrate the applications of these data structures in everyday practice.

Author of the chapter is **Mihail Valkov** and editors are Tsvyatko Konov and Svetlin Nakov. The content of the entire chapter is partially based on the work of Vladimir Tsanev (Tsachev) from the book "Introduction to Programming with Java". Translation to English: by George Mitev and George K. Georgiev (edited by martin Gebov and Ivaylo Dyankov).

Chapter 19: Data Structures and Algorithm Complexity

In the chapter "Data Structures and Algorithm Complexity", we will compare the **data structures** we have learned so far based on their **performance for basic operations** (addition, searching, deletion, etc.). We will give recommendations for the most appropriate data structures in certain cases. We will explain when it is preferable to use a **hash table**, an **array**, a **dynamic array**, a **set** implemented by a **hash table** or a **balanced tree**. There is an implementation in the .NET Framework for every one of these structures. We only have to learn how to decide when to use a particular data structure, so that we can write efficient and reliable source code.

Authors of the chapter are **Nikolay Nedyalkov** and **Svetlin Nakov**; editor is Veselin Kolev. The content of the entire chapter is based on the work of Svetlin Nakov and Nikolay Nedyalkov from the book "Introduction to Programming with Java". Translation to English: by George Halachev and Tihomir Iliev (edited by Martin Yankov).

Chapter 20: Object-Oriented Programming Principles

In the chapter "Object-Oriented Programming Principles", we will familiarize ourselves with the principles of object-oriented programming (OOP): class **inheritance**, **interfaces** implementation, data and behavior **abstraction**, data **encapsulation** and hiding implementation details, **polymorphism** and virtual methods. We will explain in detail the principles of **cohesion** and **coupling**. We will also briefly outline object-oriented modeling and object model creation based on a specific business problem and will get to know **UML** and its role in **object oriented modeling**. Finally, we will briefly discuss design patterns and provide examples for design patterns commonly used in practice.

Author of the chapter is **Mihail Stoynov** and editor is Mihail Valkov. The content of the entire chapter is based on the work of Mihail Stoynov from the book "Introduction to Programming with Java". Translation to English: by Vasya Stankova and Momchil Rogelov (edited by Ivan Nenchovski).

Chapter 21: High-Quality Programming Code

In the chapter "High-Quality Programming Code", we will take a look at the basic rules for **writing high-quality programming code**. We will focus on **naming** conventions for program elements (variables, methods, classes and others), **formatting** and code layout guidelines, best practices for creating **high-quality classes and methods**, and the principles of high-quality code documentation. Many examples of high-quality and low-quality code will be given. In the course of work, it will be explained how to use an integrated development environment, in order to automate some operations like **formatting** and **refactoring** existing code, when it is necessary. **Unit testing** as an industrial method to automated testing will also be discussed.

Authors of the chapter are **Mihail Stoynov** and **Svetlin Nakov**. Editor is Pavel Donchev. The content of the entire chapter is partially based on the work of Mihail Stoynov, Svetlin Nakov and Nikolay Vasilev from the book "Introduction to Programming with Java". Translation to English: by Blagovest Buyukliev (edited by Dyanko Petkov, Mihail Stoynov and Martin Yankov).

Chapter 22: Lambda Expressions and LINQ

In the chapter "Lambda Expressions and LINQ", we will introduce some of the more sophisticated capabilities of C#. To be more specific, we will pay special attention to clarifying how to make queries to collections using **lambda expressions** and **LINQ**. We will explain how to add functionality to already created classes, using **extension methods**. We will familiarize ourselves with **anonymous types** and briefly describe their nature and usage. We will also discuss lambda expressions and show in practice how most of the built-in lambda functions work. Afterwards we will dive into the LINQ's syntax, which is part of C#. We will learn what it is, how it works, and what queries we can make using it. Finally, we will discuss the keywords in LINQ, their meaning and we will demonstrate their capabilities with a lot of examples.

Author of the chapter is **Nikolay Kostov** and editor is Veselin Kolev. Translation to English: by Nikolay Kostov (edited by Zhasmina Stoyanova and Mihail Stoynov).

Chapter 23: Methodology of Problem Solving

In the chapter "Methodology of Problem Solving", we will discuss an advisable **approach for solving programming problems** and we will illustrate it with concrete examples. We will discuss the engineering principles we should follow when solving problems (that largely apply to problems in math, physics and other disciplines) and we will show them in action. We will describe the steps we must go through while we solve a few sample problems and demonstrate the mistakes that can be made, if we do not follow these steps. We will consider some important **steps of problem solving** (such as **testing**) that are usually skipped.

Author of the chapter is **Svetlin Nakov** and editor is Veselin Georgiev. The content of the whole chapter is entirely based on the work of Svetlin Nakov from the book "Introduction to Programming with Java". Translation to English: by Ventsi Shterev and Martin Radev (edited by Tihomir Iliev and Nedjaty Mehmed).

Chapters 24, 25, 26: Sample Programming Exam

In the chapters "Sample Programming Exam (Topic #1, Topic #2 and Topic #3)", we will look at the problem descriptions of **nine sample problems** from three sample **programming exams** and we will propose solutions to them. In the course of solving them, we will put into practice the methodology described in the chapter "Methodology of Problem Solving".

Authors of the chapters are **Stefan Staev**, **Yosif Yosifov** and **Svetlin Nakov** respectively; their respective editors are Radoslav Todorov, Radoslav Ivanov and Teodor Stoev. The contents of these chapters are largely based on the work of Stefan Staev, Svetlin Nakov, Radoslav Ivanov and Teodor Stoev from the book "Introduction to Programming with Java". Translation to English: by Stanislav Vladimirov, Ivaylo Gergov, Ivan Nenchovski and Ivaylo Gergov (edited by Dyanko Petkov, Vladimir Tsenev and Veselina Raykova).

Chapters 28: Conclusion

In the conclusion we give further instruction how to proceed with your **development as a skillful software engineer** after this book. We explain the free courses at Telerik Software Academy – the largest training center for software development professionals in Bulgaria – how to apply, what you will learn, how to choose a career path and we mention few other resources.

Author of the chapter is **Svetlin Nakov**. Translation to English: by Ivan Nenchovski (edited by Svetlin Nakov).

History: How Did This Book Come to Be?

Often in our teaching practice students ask us from which **book to start learning how to program**. There are enthusiastic young people who want to learn programming, but don't know what to begin with. Unfortunately, it's hard to recommend a good book. We can come up with many books concerning C#, but none of them teaches programming. Indeed **there aren't many books that teach the concepts of computer programming, algorithmic thinking and data structures.** Certainly, there are books for beginners that teach the C# programming language, but those rarely cover the fundamentals of programming. There are some good books on programming, but most of them are now outdated and teach languages and technologies that have become obsolete in the process of evolution. There are several such books regarding C and Pascal, but not C# or Java. Considering all aspects, it is **hard to come up with a good book** which we could highly recommend to anyone who wants to pick up programming from scratch.

At one point, the **lack of good books on programming for beginners** drove the project leader, Svetlin Nakov, to gather a panel of authors set to finally write such a book. We decided we could help many young people to take up programming seriously by sharing our knowledge and inspiring them.

The Origins of This Book

This book is actually an adaptation to C# of the free Bulgarian book "**Introduction to Programming with Java**", with some additional content added, many bug fixes and small improvements, translated later into English.

Svetlin Nakov teaches computer programing, data structures, algorithms and software technologies since 2000. He is an author and co-author of several **courses** in fundamentals of programming taught at Sofia University

(the most prestigious Bulgarian university at this time). Nakov (with colleagues) teaches programming and software development in the Faculty of Mathematics and Informatics (FMI) at Sofia University for few years and later creates his own company for training software engineers. In 2005, he gathers and leads a team of volunteers who creates a solid **curriculum on fundamentals of programming and data structures** (in C#) with presentation slides and many examples, demonstrations and homework assignments. These teaching materials are the first very early outline of the content in this book. Later this curriculum evolves and is translated to Java and serves as a base for the Java version of this book. Later the Java book is translated to C# and after its great success in Bulgaria (thousands paper copies sold and 50,000 downloads) it is translated from Bulgarian to English.

The Java Programming Fundamentals Book

In mid-2008, Svetlin Nakov is inspired to create a book on Java programming, covering his **"Introduction to Programming" course** in the National Academy for Software Development (a private training center in Bulgaria, founded by Svetlin Nakov). He and a group of authors outline the work that needs to be done and the subjects that need to be covered and work begins, with everyone working **voluntarily, without any direct profit**. Through delays, pitfalls and improvements, the Java book finally comes out in January of 2009. It is available both on its website www.introprogramming.info for free, and in a paper edition.

The C# Programming Fundamentals Book

The interest towards the "Introduction to Programming with Java" book is huge (for Bulgaria). In late 2009, the project to **"translate" the book to C#** begins, under the title "**Introduction to Programming with C#**". Again, a large number of authors, both new and from the Java book group, gather and begin working. The task seems easier, but turns out to be time-consuming. About half a year later, the "preview" edition of the book is completed – with some mistakes and incorrect content. Another year passes as all of the text and examples are improved, and new content is added. In the summer of 2011, **the C# book is released**. Its official website stays the same as the Java book (www.introprogramming.info). A paper version of the book is also released and sold, with a price covering only the expenses of its printing.

Both books are open-source and their repositories are available at Google Code: code.google.com/p/introcsharpbook, code.google.com/p/introjavabook.

The Translation of the C# Book: from Bulgarian to English

In late 2011, following the great success of "Introduction to Programming with C#", a project to **translate the book to English** started. Large group of **volunteers** began work on the translation – each of them with good programming skills. The book you are reading is the result of the successful

translation, review and completion of the original C# Bulgarian book. The most effort in the translation was given by the leading author **Svetlin Nakov**.

Some of the authors have ideas to make yet another adaptation of the book – this time **for C++**. As of now, these ideas are still foggy. We hope they will become a reality one day, but we can't promise anything yet.

Bulgaria? Bulgarian Authors? Is This True?

Bulgaria is a **country in Europe**, part of the **European Union**, just like Germany and France. Did you know this? Bulgaria has very solid traditions in computer programming and technologies.

The main inventor of the technology behind the modern digital computers is the famous computer engineer **John Atanasoff** and he is **50% Bulgarian** (see en.wikipedia.org/wiki/John_Vincent_Atanasoff).

Bulgaria is the founder of the **International Olympiad in Informatics (IOI)**. The first IOI was organized and held in 1980 in Pravetz, Bulgaria (see en.wikipedia.org/wiki/International_Olympiad_in_Informatics).

In 2011 Bulgaria was ranked **#3 in the world by Internet speed** (see http://mashable.com/2011/09/21/fastest-download-speeds-infographic).

The world's leading component vendor for the Microsoft ecosystem is a Bulgarian company called **Telerik** (www.telerik.com) and almost all of its products are developed in Bulgaria. The world's leading software product for 3D rendering (V-Ray), used in most Hollywood movies and by most automotive producers, is invented and developed in Bulgaria by another Bulgarian company – **Chaos Group** (www.chaosgroup.com). A Bulgarian company **Datecs** designed and produces the barcode scanner with credit card swipe for Apple iPhone / iPad / iPod devices used in all Apple stores. Large international software companies like **SAP**, **VMware**, **HP**, **Cisco**, **Siemens** and **CSC** have large development centers in Sofia with thousands developers.

Bulgarian software engineers can be found in every major software company in the software industry like Microsoft, Google, Oracle, SAP, Facebook, Apple, IBM, Cisco, Siemens, VMware, HP, Adobe, Nokia, Ericsson, Autodesk, etc.

We, the authors, editors and translators of this book are all **proud Bulgarian software developers** – some living in Bulgaria, others abroad. We are happy to be part of the global software industry and to help beginners over the world to learn computer programming and become skillful software engineers. We are supporters of **the culture of free education** (like Coursera, edX, Udacity and Khan Academy), free education for everyone and everywhere. We are happy to share our knowledge, skills and expertise and **sharing is part of our culture**.

Authors and Contributors

This book is **written by volunteer developers from Bulgaria** who want to share their knowledge and skills about computer programming. They have

worked for months (some for years) for free to help the community to **learn programming**, data structures and algorithms in an easy and efficient way: through this book and the presentations and video tutorials coming with it.

Over 70 people contributed to the project: authors, editors, translators, etc.

The Panel of Authors

The panel of authors of both the old, the new and the translated to English book is indeed the main drivers behind this book's existence. Writing content of this size and quality is a serious task demanding a lot of time.

The idea of having so many authors participating has been well tested, since a few other books have already been written in a similar manner (e.g. "Programming for the .NET Framework" – parts 1 and 2). Although **all chapters from the book are written by different authors**, they adhere to the same style and possess the same high quality of content (even though it might differ a little in some chapters). The text is well structured, has many titles and subtitles, contains many appropriate examples, follows a good manner of expression and is uniformly formatted.

The team that wrote this book is made up of people who are strongly interested in programming and would like to **voluntarily share their knowledge** by participating in writing one or more of the chapters. The best part is that all authors, co-authors and editors in the team working on the book are working programmers with **hands-on experience**, which means that the reader will receive knowledge, a collection of best practices and advice by people with an active career in the software industry.

The participants in the project made their contribution voluntarily, without material or any other direct compensation, because they **supported the idea of writing a good book for novice programmers** and because they strongly wanted to help their future colleagues get into programming quickly.

What follows is a brief presentation of the **authors of the book** "Introduction to Programming with C#" (in an alphabetical order). The original authors of the corresponding chapters from the book "Introduction to Programming with Java" are mentioned accordingly, since their contributions to some chapters are greater than those authors who adapted the text and examples to C# afterwards.

Dilyan Dimitrov

Dilyan Dimitrov is a certified software **developer** with professional experience in building mid-size and large web-based systems with the **.NET Framework**. His interests include development of both web and desktop applications using Microsoft's latest technologies. He graduated from the Sofia University "St. Kliment Ohridski" where he majored in "Informatics" at the Faculty of Mathematics and Informatics. . He can be reached at dimitrov.dilqn@gmail.com or you can visit his personal blog at http://dilyandimitrov.blogspot.com.

Hristo Germanov

Hristo Germanov is a **software engineer**, whose interests are related mainly to **.NET technologies**. Architecture and design of web based systems, algorithms and modern standards for quality code are also his passion. He has participated in developing both small and large web-based and desktop-based applications. He likes **challenging problems** and projects that require strong logical thinking. He graduated from the Omega College in Plovdiv with a degree in "Computer Networks". He specialized for a "Core .NET Developer" at the National Academy for Software Development in Sofia.

You can contact him by e-mail at: hristo.germanov@gmail.com.

Iliyan Murdanliev

Iliyan Murdanliev is a **software developer** at NearSoft (www.nearsoft.eu). He currently pursues a master's degree in "Computer Technologies and Applied Programming" at the Technical University of Sofia. He has a bachelor's degree in "Applied Mathematics" from the same university. He has graduated from an English language high school.

Iliyan has participated in significant projects and in the development of front-end visualization, as well as back-end logic. He has prepared and conducted **trainings in C#** and other programming languages and technologies. Iliyan's interests lie in the field of cutting-edge technologies in .NET, Windows Forms and Web-based technologies, design patterns, algorithms and software engineering. He likes **out-of-the-box projects** that require not only knowledge, but also logical thinking.

His personal blog is available at: http://imurdanliev.wordpress.com. He can be reached by e-mail: i.murdanliev@gmail.com.

Mihail Stoynov

Mihail Stoynov has a master's degree in "Economics and Management" from the Sofia University "St. Kliment Ohridski". He has obtained his bachelor's degree in "Informatics" also from Sofia University.

Mihail is a **professional software developer**, consultant and instructor with many years of experience. For the last few years he is an honorary instructor at the Faculty of Mathematics and Informatics and has **delivers lectures** in the "Networks Theory", "Programming for the .NET Framework", "Java Web Applications Development", "Design Patterns" and "High Quality Programming Code" courses. He has also been an **instructor** at New Bulgarian University.

He is an author of a **number of articles and publications** and a **speaker at many conferences** and seminars in the field of software technologies and information security. Mihail is a co-author of the books "Programming for the .NET Framework" and "Introduction to Programming with Java". He has participated in Microsoft's MSDN Academic Alliance and is a lecturer at the Microsoft Academic Days.

Mihail has **led IT courses** in Bulgaria and abroad. He was a lecturer in the "Java", "Java EE", "SOA" and "Spring Framework" courses at the National Academy for Software Development.

Mihail has worked at the international offices of Siemens, HP and EDS in the Netherlands and Germany, where he has gained a lot of experience in the art of software, as well as in the **quality programming**, by taking part in the development of large software projects. His interests encompass software architectures and design development, B2B integration of various information systems, business processes optimization and software systems mainly for the **Java** and **.NET platforms**. Mihail has participated in dozens of software projects and has extensive experience in web applications and services, distributed systems, relational databases and ORM technologies, as well as management of projects and software development teams.

His personal blog is available at: http://mihail.stoynov.com. His twitter account is available at: https://twitter.com/mihailstoynov.

Mihail Valkov

Mihail Valkov has been a **software developer** since 2000. Throughout the years, he has faced numerous technologies and software development platforms, some of which are MS .NET, ASP, Delphi. Mihail has been developing software at Telerik (www.telerik.com) ever since 2004. There he co-develops a number of components targeting ASP.NET, Windows Forms, Silverlight and WPF. In the last few years, Mihail has been **leading some of the best progressing teams** in the company, and currently develops an online Word-like rich text editor.

He can be reached at: m.valkov@gmail.com.

His blog is at: http://blogs.telerik.com/mihailvalkov/. His twitter account is available at: https://twitter.com/mvalkov.

Mira Bivas

Mira Bivas is an enthusiastic **young programmer** in one of Telerik's ASP.NET teams (www.telerik.com). She is a student at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski", where she majors in "Applied Mathematics". Mira has completed the "Intro C#" and "Core .NET" courses at the National Academy for Software Development (NASD).

She can be reached by e-mail: mira.bivas@gmail.com.

Nikolay Kostov

Nikolay Kostov works as a **senior software developer** and **trainer** at Telerik's "Technical Training" department (http://academy.telerik.com). He is involved deeply with Telerik Academy's **trainings** and the courses organized by Telerik. He currently majors in "Computer Science" at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski".

Nikolay has participated in a number of high school and college student **Olympiads and contests in computer science**, throughout many years. He is a two-time champion in the project categories "Desktop Applications" and "Web Applications" at the Bulgarian National Olympiad in Information Technologies (NOIT). He has rich experience in designing and developing Web applications, algorithmic programming and processing large amounts of data.

His main interests lie in developing software applications, data structures, everything related to **.NET technologies**, web applications security, data processing automation, web crawlers, single page applications and others.

Nikolay's personal blog can be found at: http://nikolay.it.

Nikolay Nedyalkov

Nikolay Nedyalkov is the chairman of The Association for Information Security, technical director of the eBG.bg's electronic payments and services portal and business consultant at other companies. Nikolay is a professional **software developer**, **consultant** and **instructor** with many years of experience. He has authored a number of articles and **publications** and has lectured at many conferences and seminars in the field of software technologies and information security. His experience as an **instructor** ranges from assisting in "Data Structures in Programming", "Object-oriented Programming with C++" and "Visual C++" to lecturing at the "Network Security", "Secure Code", "Web Development with Java", "Creating High Quality Code", "Programming for the .NET platform" and "Applications Development with Java" courses. Nikolay's interests are focused on creating and managing information and communications solutions, modeling and managing **business processes** in large-size organizations and state administration. Nikolay has a bachelor's and a master's degree from the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski". As a high school student he was a **programming contestant** throughout many years and received a number of accolades.

His personal website is located at: http://www.nedyalkov.com.

Nikolay Vasilev

Nikolay Vasilev is a **professional software developer**, an **instructor** and a participant in many open source projects.

He holds a master's degree in "Software Engineering and Artificial Intelligence" from University of Malaga (Spain) and is currently pursuing a master's degree in "Mathematical Physics Equations and Their Applications" at Sofia University (Bulgaria). He obtained his bachelor's degree in "Mathematics and Informatics" from Sofia University.

In the period 2002-2005, he was **instructor** in the classes of "Introduction in Programming with Java" and "Data Structures and Programming with Java" at Sofia University.

Nikolay is a **co-author of the books** "Introduction in Programming with Java" and "Introduction in Programming with C#" and also one of the initiators, organizers and co-authors of a project for creating an open source book in Bulgarian, dedicated to the classical (GoF) design patterns in the software engineering. He is one of the organizers and lecturers of the "Bulgarian Java User Group".

Nikolay is a **certified software developer** with nearly 10 years of expertise in development of Java enterprise applications, gained in international companies. He participated in large-size systems development from various domains like e-commerce, banking, visual simulators for nuclear plant sub-systems, VOD systems, etc.; using cutting-edge technologies and applying the best up-to-date design and development methodologies and practices. His interests span across various areas such as software engineering and artificial intelligence, fluid mechanics, project management and scientific research.

Nikolay Vasilev's personal blog is available at http://blog.nvasilev.com.

Pavel Donchev

Pavel Donchev is a **programmer** at Telerik (www.telerik.com), where he develops web applications mostly for the company internal purposes. He takes extramural courses in "Theoretical Physics" at the Sofia University "St. Kliment Ohridski". He was engaged in developing **Desktop and Web Applications** for various business sectors – mortgage credits, online stores, automation and Web UML diagrams. His interests lie mainly in the sphere of process automation using Microsoft technologies.

His personal blog is located at: http://donchevp.blogspot.com.

Pavlina Hadjieva

Pavlina Hadjieva is a **senior enterprise support officer** and team lead at Telerik (www.telerik.com). She currently pursues a master's degree in "Distributed Systems and Mobile Technologies" at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski". She obtained her bachelor's degree in "Chemistry and Computer Science" also from Sofia University.

Her professional interests are oriented towards web technologies, in particular **ASP.NET**, as well as the complete **development cycle** of .NET Framework applications.

You can contact Pavlina Hadjieva by e-mail: pavlina.hadjieva@gmail.com.

Radoslav Ivanov

Radoslav Ivanov is an **experienced software engineer**, consultant and **trainer** with several years of professional experience in wide range of technologies and programming languages. He has solid practical and theoretical background in computer science and excellent **writing** and **lecturing skills**.

Radoslav has a bachelor's degree in "Informatics" and master's degrees in "Software Engineering" and "E-learning" from the Sofia University "St. Kliment Ohridski". For several years he has been an **honorary instructor** at the Faculty of Mathematics and Informatics where he was teaching courses in "Design Patterns in C#", "Programming for the .NET Framework", "Java Web Applications Development" and "Java EE Development".

He is a co-author of the books "Programming for the .NET Framework" and "Introduction to Programming with Java".

His professional interests include data warehousing, security, cloud computing, Java technologies, the .NET platform, software architecture and design and project management.

Radoslav's twitter account is available at: https://twitter.com/radoslavi.

Radoslav Kirilov

Radoslav Kirilov is a **senior software developer** and **team leader** at Telerik (www.telerik.com). He graduated from the Technical University of Sofia with a major in "Computer Systems and Technologies". . His professional interests are oriented towards web technologies, particularly **ASP.NET**, and the complete development cycle of .NET Framework-based applications. Radoslav is an experienced lecturer who has taken part in putting through, as well as **creating study materials** (presentations, examples, exercises) for the National Academy for Software Development (NASD). Radoslav is a member of the **instructors' team** of the "High Quality Programming Code" course that started in 2010 at the Technical University of Sofia and at the Sofia University "St. Kliment Ohridski".

He has been maintaining a tech blog since 2009 located at: radoslavkirilov.blogspot.com. You can contact Radoslav by e-mail at: radoslav.pkirilov@gmail.com.

Radoslav Todorov

Radoslav Todorov is a **software developer** who obtained his bachelor's degree from the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski" (www.fmi.uni-sofia.bg). He received his master's degree in the field of computer science from the Technical University of Denmark in Lyngby, Denmark (http://www.dtu.dk).

Radoslav has been conducting courses as an **instructor-assistant** at the IT University of Copenhagen in Denmark (http://www.itu.dk) and participating in the research activity of university projects ever since he received his masters' education. He has rich experience in designing, developing and maintaining large software products for various companies. He gained **working experience** at several companies in Bulgaria. At present, he works as a software engineer for Canon Handy Terminal Solutions Europe in Denmark (www.canon-europe.com/Handy_Terminal_Solutions).

Radoslav’s interests are oriented towards software technologies for high-level programming languages, as well as products integrating complete hardware and software solutions in the industrial and private sectors.

You can contact Radoslav by e-mail: radoslav_todorov@hotmail.com.

Stanislav Zlatinov

Stanislav Zlatinov is a **software developer** with professional experience in web and desktop applications development based on the **.NET** and **Java** platforms.

He has a master’s degree in "Computer Multimedia" from the "St. Cyril and St. Methodius" University of Veliko Tarnovo.

His personal blog is located at: http://encryptedshadow.blogspot.com.

Stefan Staev

Stefan Staev is a **software developer** who is occupied with building web based systems using the .NET platform. His professional interests are related to the latest **.NET technologies**, design patterns and databases. He is a member of the authors' team of the book "Introduction to Programming with Java".

Stefan currently majors in "Informatics" at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski". He is a "**Core .NET Developer"** graduate from the National Academy for Software Development.

You can contact him by e-mail: stefosv@gmail.com. His Twitter micro blog is located at: http://twitter.com/stefanstaev.

Svetlin Nakov

Svetlin Nakov is the head of the "**Technical Training**" department at Telerik Corp. where he manages the project for free training of software engineers **Telerik Software Academy** (http://academy.telerik.com) as well as all other connected courses and training initiatives, such as Telerik School Academy, Telerik Algo Academy, Telerik Kids Academy. He is the founder of the Software University open-education project.

He has achieved a bachelor’s degree in "Computer Science" and a master’s degree in "Distributed Systems and Mobile Technologies" at the Sofia University "St. Kliment Ohridski". Later he obtained a **Ph.D. in "Computer Science"** after defending a thesis in the field of "Computational Linguistics" before the Higher Attestation Commission of the Bulgarian Academy of Sciences (BAS).

His interests encompass software architectures development, the **.NET platform**, web applications, databases, Java technologies, training software specialists, information security, technological entrepreneurship and managing software development projects and teams.

Svetlin Nakov has nearly **20 years of experience as a software engineer**, programmer, **instructor** and consultant, moving from Assembler, Basic and Pascal through C and C++ to PHP, JavaScript, Java and C#. He was involved as a software engineer, consultant and manager of teams in dozens of projects for developing information systems, web applications, database management systems, business applications, ERP systems, cryptographic modules and trainings of software engineers. At the age of 24, he founded his first **software company for training software engineers**, which was acquired 5 years later by Telerik.

Svetlin has extensive experience in creating **study materials**, preparing and conducting **trainings** in programming and modern software technologies, gathered during his practice as an instructor. For many years now, he has been an **honored instructor** at the Faculty of Mathematics and Informatics at the **Sofia University "St. Kliment Ohridski"** (FMI at SU), at the **New Bulgarian University** (NBU) and at the **Technical University of Sofia** (TU-Sofia), where he held **courses** in "Design and Analysis of Computer Algorithms", "Internet and Web Programming with Java", "Network Security", "Programming for the .NET Framework", "Developing Java Web Applications", "Design Patterns", "High Quality Programming Code", "Developing Web Applications with the .NET Framework and ASP.NET", "Developing Java and Java EE Applications", "Web Front-End Development" and many others (see http://www.nakov.com/courses/).

Svetlin has dozens of **scientific and technical articles** focused on software development in both Bulgarian and foreign publications and is the lead author of the **books** "Programming for the .NET Framework (vol. 1 & 2)", "Introduction to Programming with Java", "Introduction to Programming with C#", "Internet Development with Java" and "Java for Digitally Signing Web Documents". He is a **regular speaker** at technical conferences, trainings and seminars and up to now has held hundreds of technical lectures at various technological events in Bulgaria and abroad.

As a high school and a college student, Svetlin was champion in tens of national contests in programming and was awarded with **4 medals** at International Olympiads in Informatics (IOI).

In 2003, he received the "John Atanasoff" award by the EVRIKA Foundation. In 2004, he was **awarded by the Bulgarian President** with the "John Atanasoff" award for his contribution to the development of the information technologies and the information society.

He is one of the founders of the **Bulgarian Association of Software Developers** (www.devbg.org) and its present chairman.

Apart from computer programming, Svetlin Nakov is founder of **NLP Club Bulgaria** (http://nlpclub.devbg.org), a community of NLP (neuro-linguistic programming) practitioners and successful people who are looking for personal development and knowledge sharing. The goal for Svetlin is to add **soft skills** and **personal development** to his students at the Software academy in addition to the profession and job positions they gain.

The personal website and blog of Svetlin Nakov is: http://www.nakov.com. His story of life is published at http://www.nakov.com/blog/2011/09/24/.

Teodor Bozhikov

Teodor Bozhikov is a **senior software developer** and **team leader** at Telerik (www.telerik.com). He completed his master's degree in "Computer Systems and Technologies" at the Technical University of Varna. Besides his background as a WPF and Silverlight programmer, he has achieved expertise in developing ASP.NET web applications. He was involved briefly in the development of private websites. Within the ICenters project, he took part in building and maintaining of a local area network for public use at the Festival and Congressional Center in Varna. He has held **courses** in computer literacy and computer networks basics.

Teodor's professional interests include web and desktop application development technologies, architecture and design patterns, networks and all kinds of new technologies.

You can contact Teodor by e-mail: t_bozhikov@yahoo.com. His Twitter micro blog is located at: http://twitter.com/tbozhikov.

Teodor Stoev

Teodor Stoev has a bachelor's and a master's degree in "Informatics" from the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski". At Sofia University, he mastered in "Software Technologies". He currently attends a master's program in "Computer Science" at the Saarland University (Saarbrücken, Germany).

Teodor is a **software designer and developer** with many years' experience. He has participated in creating financial and insurance software systems, a number of web applications and corporate websites. He was actively involved in the development of the TENCompetence project of the European Commission. He is a **co-author** of the **book** "Introduction to Programming with Java".

His professional interests lie in the field of object-oriented analysis, modeling and building of software applications, web technologies and, in particular, building rich internet applications (RIA). He has an extensive background in **algorithmic programming**: he has competed at a number of national high school and collegiate computer science contests.

His personal website is available at: http://www.teodorstoev.com.

You can contact Teodor by e-mail: teodor.stoev@gmail.com.

Tsvyatko Konov

Tsvyatko Konov is a **senior software developer** and **instructor** with varied interests and experience. He is competent in fields such as systems integration, building software architectures, developing systems with a number of technologies, such as **.NET Framework**, ASP.NET, Silverlight,

WPF, WCF, RIA, MS SQL Server, Oracle, MySQL, PostgreSQL and PHP. His experience as an instructor includes a large variety of **courses** – courses for beginners and experts in .NET technologies, as well as specialized courses in individual technologies, such as ASP.NET, Oracle, .NET Compact Framework, "High Quality Programming Code" and others. Tsvyatko was part of the **authors' team** of the **book** "Introduction to Programming with Java". His professional interests include web-based and desktop-based technologies, client-oriented web technologies, databases and design patterns.

Tsvyatko Konov has a technical blog: http://www.konov.me.

Veselin Georgiev

Veselin Georgiev is a **co-founder of Lead IT** (www.leadittraining.com) and software developer at Abilitics (www.abilitics.com). He has a master's degree in "E-Business and E-Governance" at the Sofia University "St. Kliment Ohridski", after obtaining a bachelor's degree in "Informatics" from the same university.

Veselin is a **Microsoft Certified Trainer** and Microsoft Certified Professional Developer. He lectured at the Microsoft Tech Days conferences in 2011 and 2009, and also takes part as an instructor in various courses at Sofia University. He is an **experienced lecturer** who has trained software specialists for working practical jobs in the IT industry.

His professional interests are oriented towards training, SharePoint and software architectures. He can be reached at veselin.vgeorgiev@gmail.com.

Veselin Kolev

Veselin "Vesko" Kolev is a **leading software engineer** with many years' professional experience. He has worked at various companies where he **managed teams** and the development of many different software projects. As a high school student, he participated in a number of competitions in the fields of mathematics, computer science and information technology, where he finished in prestigious places. He currently majors in "Computer Science" at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski".

Vesko is an **experienced lecturer** who has worked on training software specialists for practical jobs in the IT industry. He is an **instructor** at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski" where he conducts courses in "Modern Java Technologies" and "High Quality Programming Code". He has delivered similar lectures at the Technical University of Sofia.

Vesko's main interests include software projects design, development of software systems, **.NET** and Java technologies, Win32 programming (C/C++), software architectures, design patterns, **algorithms**, databases, team and software projects management, specialists training. The projects he has worked on include large web based systems, mobile applications, OCR,

automated translation systems, economic software and many others. Vesko is a **co-author** of the **book** "Introduction to Programming with Java".

Vesko works on the development of Silverlight and WPF based applications at Telerik (www.telerik.com). He shares parts of his day-to-day experiences online on his personal blog at http://veskokolev.blogspot.com.

Yordan Pavlov

Yordan Pavlov has a bachelor's and a master's degree in "Computer Systems and Technologies" from the Technical University of Sofia. He is a **software developer** at Telerik (www.telerik.com) with an extensive background in software components development.

His interests lie mainly in the following fields: object-oriented design, design patterns, **high-quality software development**, geographic information systems (GIS), parallel processing and high performance computing, artificial intelligence, teams' management.

Yordan **won the Imagine Cup 2008** finals in Bulgaria in the Software Design category, as well as the world finals in Paris, where he won Microsoft's prestigious "The Engineering Excellence Achievement Award". He has worked with Microsoft engineers at the company headquarters in Redmond, USA, where he has gathered useful knowledge and experience in the development of complex software systems.

Yordan has also received a **golden mark** for "Contributions to the Innovation and Information Youth Society". He has taken part in many **contests** and **Olympiads** in programming and informatics.

Yordan's personal blog can be found at http://yordanpavlov.blogspot.com. He can be reached by e-mail: iordanpavlov@gmail.com.

Yosif Yosifov

Yosif Yosifov is a **senior software developer** at Telerik (www.telerik.com). His interests consist mainly of **.NET technologies**, design patterns and computer **algorithms**. He has participated in numerous **contests** and **Olympiads** in programming and informatics. He currently pursues a bachelor's degree in "Computer Science" at the Faculty of Mathematics and Informatics at the Sofia University "St. Kliment Ohridski".

Yosif's personal blog can be found at http://yyosifov.blogspot.com. He can be reached by e-mail: cypressx@gmail.com.

The Java Book Authors

This C# fundamentals programming book is based on its **original Java version**, the book "Introduction to Programming with Java". Thanks to the original Java book authors for their work. They have significant contribution to almost all chapters of the book. Some chapters are entirely based on their

work, some partially, but in all cases their original work is the primary origin of this book:

- **Boris Valkov**
- **Danail Aleksiev**
- **Hristo Todorov**
- **Lachezar Bozhkov**
- **Luchesar Cekov**
- **Marin Georgiev**
- **Mario Peshev**
- **Mariyan Nenchev**
- **Mihail Stoynov**
- **Nikolay Nedyalkov**
- **Nikolay Vasilev**
- **Petar Velev**
- **Radoslav Ivanov**
- **Rumyana Topalska**
- **Stefan Staev**
- **Svetlin Nakov**
- **Teodor Stoev**
- **Vesselin Kolev**
- **Vladimir Tsanev**
- **Yosif Yosifov**

The Editors

Apart from the authors, a **significant contribution** to the making of this book was made by the editors who voluntarily took part in reviewing the text and the examples and fixing errors and other problems:

- **Dilyan Dimitrov**
- **Doncho Minkov**
- **Hristo Radkov**
- **Iliyan Murdanliev**
- **Marin Georgiev**
- **Mihail Stoynov**
- **Mihail Valkov**
- **Mira Bivas**
- **Nikolay Kostov**
- **Nikolay Vasilev**
- **Pavel Donchev**
- **Radoslav Ivanov**
- **Radoslav Kirilov**
- **Radoslav Todorov**
- **Stanislav Zlatinov**
- **Stefan Staev**
- **Svetlin Nakov**
- **Teodor Bozhikov**
- **Tsvyatko Konov**
- **Veselin Georgiev**
- **Veselin Kolev**
- **Yosif Yosifov**

The Translators

This book would have remained only in Bulgarian for many years if these guys hadn't volunteered **to translate it in English**:

- **Angel Angelov**
- **Atanas Valchev**
- **Blagovest Buyukliev**
- **Boyan Dimitrov**
- **Dimitar Bonev**
- **Doroteya Agayna**
- **Dyanko Petkov**
- **Franz Fischbach**
- **George Halachev**
- **George K. Georgiev**
- **George S. Georgiev**
- **Georgi Mitev**
- **Georgi Todorov**
- **Georgi Vaklinov**
- **Hristo Radkov**
- **Ivan Nenchovski**
- **Ivaylo Dyankov**
- **Ivaylo Gergov**
- **Zhasmina Stoyanova**
- **Kristian Dimitrov**
- **Lora Borisova**
- **Martin Gebov**
- **Martin Radev**
- **Martin Yankov**
- **Momchil Rogelov**
- **Nedjaty Mehmed**
- **Nencho Nenchev**
- **Nikolay Angelov**
- **Nikolay Kostov**
- **Pavel Benov**
- **Radoslav Todorov**

- **Stanislav Vladimirov**
- **Svetlin Nakov**
- **Teodor Rusev**
- **Tihomir Iliev**
- **Todor Mitev**
- **Vasya Stankova**
- **Ventsi Shterev**
- **Vesselin Georgiev**
- **Vesselina Raikova**
- **Vladimir Amiorkov**
- **Vladimir Tsenev**
- **Yoan Krumov**
- **Zhelyazko Dimitrov**

Many thanks to **George S. Georgiev** who was seriously involved in the translation process and edited the translated text for most of the chapters.

Other Contributors

The authors would also like to thank **Kristina Nikolova** for her efforts in working out the book's cover design. Big thanks to **Viktor Ivanov** and **Peter Nikov** for their work on the project's web site. Big thanks to **Ivaylo Kenov** for fixing few hundreds bugs reported in the Bulgarian edition of the book. Thanks to **Ina Dobrilova** and **Aneliya Stoyanova** for the proofreading of the first few chapters and their contribution to the marketing of the book. Many thanks to **Hristo Radkov** who is proficient in English (lives and works in London for many years) and who edited and corrected the translation of the first few chapters.

The Book Is Free of Charge!

The present book is distributed **absolutely free of charge** in an electronic format under a license that grants its usage for all kinds of purposes, including commercial projects. The book is also distributed in paper format for a charge, covering its printing and distribution costs without any profit.

Reviews

If you don't fully trust the authors who wrote this book, you can take inspiration from its **reviews written by leading worldwide specialists**, including software engineers at Microsoft, Google, Oracle, SAP and VMware.

Review by Nikola Mihaylov, Microsoft

Programming is an awesome thing! People have been trying for hundreds of years to make their lives easier, in order to work less. Programming allows humanity's tendency towards laziness to continue. If you are a computer freak or if you'd just like to impress others with a good website or something of yours "never-seen -before", then you are welcome. No matter if you are part of the relatively small group of "freaks" who get off on encountering a nice program or if you'd just like to fulfill yourself professionally and lead your life outside the workplace, **this book is for you**.

The fundamental concepts of a car's engine haven't changed in years – something inside it burns (gas, oil or whatever you have filled it with) and the car rolls along. Likewise, the concepts of programming haven't changed for

years. Whether you write the next video game, money management software in a bank or you program the "mind" of a new bio robot, you will use – with absolute certainty – **the concepts and the data structures described in this book**.

In this book, you will find a large part of the **programming fundamentals**. An analogical fundamental book in the automobile industry would be titled "Internal Combustion Engines".

Whatever you do, it's most important **to enjoy it**! Before you start reading this book, think of something you'd like to do as a programmer – a website, a game or some other program! While reading the book, think of how and what from the stuff you have read you would use in your program! If you find something interesting, you would learn it easily!

My first program (of which I'm proud enough to speak of in public) was simply drawing on the screen using the arrow keys of the keyboard. It took me quite some time to write it back then, but when it was done, I liked it. I wish you this: may you like everything related to programming! Have a nice reading and a successful professional fulfillment!

Nikola Mihaylov *is a software engineer at Microsoft in the team developing Visual Studio. He is the author of the website http://nokola.com and is easily "turned on" by the topic of programming; he is always ready when it's necessary to write something positive! He loves helping people with questions and a desire for programming, no matter if they are beginners or experts. When in need, contact him by e-mail: nokola@nokola.com.*

Review by Vassil Bakalov, Microsoft

"Introduction to Programming with C#" is a brave effort to not only help the reader make their first steps in programming, but also to introduce them with the programming environment and to **train for the practical tasks that occur in a programmer's day-to-day life**. The authors have found a good combination of theory – to pass over the necessary knowledge for writing and reading programming code – and practice – all kinds of problems, carefully selected to assimilate the knowledge and to form a habit in the reader to always think of the efficient solution to the problem in addition to the syntax when writing programs.

The **C# programming language** is a good choice, because it is an elegant language through which the program's representation in the computer memory is of no concern to us and we can concentrate on improving the efficiency and elegance of our program.

Up until now I haven't come across a programming book **that introduces its reader with the programming language** and **develops their problem solving skills** at the same time. I'm happy now that there is such a book and I'm sure it will be of great use to future programmers.

Vassil Bakalov *is a software engineer at Microsoft Corporation (Redmond) and a participant in the project for the first Bulgarian book for .NET:*

"Programming for the .NET Framework". His blog is located at: http://bakalov.com.

Review by Vassil Terziev, Telerik

Skimming through the book, I remembered the time, when I was making **my first steps in PHP programming**. I still remember the book I learned from – four authors, very disorganized and incoherent content and elementary examples in the chapters for experts and complicated examples in the chapters for beginners, different coding conventions and emphasis only on the platform and the language and not on how to use them efficiently for writing high quality applications.

I'm very glad that "Introduction to Programming with C#" takes an entirely different approach**. Everything is explained in an easy to understand manner**, but with the necessary profundity, and every chapter goes on to slowly extend the previous ones. As an outside bystander I was a witness of the efforts put into writing the book and I'm happy that this immense energy and desire to create a more different book truly has materialized in a subject matter of very high quality.

I strongly hope that **this book will be useful** to its readers and that it will provide them with a strong basis for finding their feet, a basis that will hook them on to a professional development in the field of computer programming and that will help them make a more painless and qualitative start.

***Vassil Terziev** is one of the founders and CEO of Telerik Corporation, leading provider of developer tools and components for .NET, HTML5 and mobile development. His blog is located at http://blogs.telerik.com/vassilterziev/. You can contact him at any time you want by e-mail: terziev@telerik.com.*

Review by Veselin Raychev, Google

Perhaps even without reading this, you'll be able to work as a software developer, but I think you'll find it much more difficult.

I have seen cases of reinventing the wheel, often times in a worse shape than the best in theory and the entire team suffers mostly from this. Everybody committed to programming must sooner or later read what **algorithm complexity** is, what a **hash table** is, what **binary search** is and what the best practices for using **design patterns** are. Why don't you start at this very moment by reading this book?

There are many books on C# and much more on programming. People would say about many of them that they are the best guides, the fastest way to get into the swing of the language. This book differs from others mainly because it will show you what you must know to achieve success and not what the twists and turns of a given programming language are. **If you find the topics covered in this book uninteresting, then software engineering might possibly not be for you.**

***Veselin Raychev** is a software engineer at Google where he works on Google Maps and Google Translate. He has previously worked at Motorola Biometrics and Metalife AG.*

*Veselin has won accolades at a number of **national and international contests** and received a **bronze medal** at the International Olympiad in Informatics (IOI) in South Korea, 2002, and a **silver medal** at the Balkan Olympiad in Informatics (BOI). He represented the Sofia University "St. Kliment Ohridski" twice at the **world finals** in computer science (ACM ICPC) and taught at several optional courses at the Faculty of Mathematics and Informatics at the University of Sofia.*

Review by Vassil Popovski, VMware

As an employee at a managing position at VMware and at Sciant before that, I often have to carry out **technical interviews for job candidates** at our company. It's surprising how many of the candidates for software engineering positions that come to us for an interview don't know **how a hash table works**, haven't heard of algorithm complexity, cannot sort an array or sort it with a complexity of $O(n^3)$. It's hard to believe the amount of self-taught programmers that haven't mastered the fundamentals of programming you'll find in this book. Many people practicing the software developer profession are not even familiar with the most basic data structures in programming and don't know how to iterate through a tree using recursion. **Read this book, so that you won't be like these people!** It is the first textbook you should start with during your training as a programmer. The fundamental knowledge of **data structures**, **algorithms** and **problem solving** will be necessary for you to build your **carrier in software engineering** successfully and, of course, to be successful at job interviews and the workplace afterwards.

If you start with creating dynamic websites using databases and AJAX without knowing what a linked list, tree or hash table is, one day you'll find out what **fundamental gaps** there are in your skill set. Do you have to make a fool of yourself at a job interview, in front of your colleagues or in front of your superior when it becomes clear that you don't know the purpose of a hash code, or how the **`List<T>`** structure works or how hard drive folders are traversed recursively?

Most programming books will teach you to write simple programs, but they won't take into consideration **the quality of the programming code**. It is a topic most authors find unimportant, but writing high quality code is a basic skill that separates the capable programmers from the mediocre ones. Throughout the years you might discover the best practices yourself, but do you have to learn by trial and error? This book will show you the right course of action the easy way – **master the basic data structures and algorithms**; learn to **think correctly**; and **write your code with high-quality**. I wish you beneficial studying.

***Vassil Popovski** is a software architect at VMware Bulgaria with more than 10 years of experience as a Java developer. At VMware Bulgaria he works on*

developing scalable Enterprise Java systems. He has previously worked as senior manager at VMware Bulgaria, as technical director at Sciant and as team leader at SAP Labs Bulgaria.

*As a high school student Vassil won awards at a number of national and international contests including a **bronze medal** at the International Olympiad in Informatics (IOI) in Setúbal, 1998, and a **bronze medal** at the Balkan Olympiad in Informatics (BOI) in Drama, Greece, 1997. As a college student, Vassil participated in a number of college contests and in the worldwide interuniversity contest in programming (ACM ICPC). During the 2001/2002 period, he **held the course** "Transaction Processing" at the Sofia University "St. Kliment Ohridski". Vassil is one of the founders of the Bulgarian Association of Software Developers (BASD).*

Review by Pavlin Dobrev, ProSyst Labs

The book "Introduction to Programming with C#" is an **excellent study material for beginners** that gives you the opportunity to master the fundamentals of programming in an easy to understand manner. It's the seventh book written under the guidance of Svetlin Nakov and just like the others, it's oriented exclusively to **gaining practical programming skills**. The subject matter includes fundamental topics such as data structures, algorithms and problem solving and that makes it intransient in technologies' development. It's filled with countless examples and practical advice for solving basic problems from a **programmer's everyday work**.

The book "Introduction to Programming with C#" represents an adaptation of the **incredibly successful book** "Introduction to Programming with Java" to the C# programming language and Microsoft's .NET Framework platform and is based on its leading author's, Svetlin Nakov, experience gained while teaching programming fundamentals – not only at the **National Academy for Software Development (NASD)** and later at **Telerik Software Academy**, but at the Faculty of Mathematics and Informatics at the **Sofia University** "St. Kliment Ohridski", at the **New Bulgarian University** and at the **Technical University of Sofia** as well.

Despite the large number of authors, all of which with differing professional and training experience, there is a clear logical connection between the separate chapters from the book. It's **clearly written**, **with detailed explanations** and **many, many examples** far from the dull academic style of most university textbooks.

Oriented towards those making their first steps in programming, the book delivers carefully, step by step, the **most important stuff** a programmer must be proficient in, in order to practice his profession – starting from variables, loops and arrays, to fundamental data structures and algorithms. The book also covers important topics like recursive algorithms, trees, graphs and hash tables. It's one of the few books that **teach a good programming style** and high-quality programming code at the same time. There is enough

thought put into the object-oriented programming principles and exceptions handling, without which modern software development is unimaginable.

The book "Introduction to Programming with C#" teaches **the most important principles and concepts in programming** in the way programmers think when solving problems in their everyday work.

This book doesn't contain everything about programming and won't make you .NET software engineers. If you want to become **really good programmer**, you need lots and lots of practice. Start from the exercises at the end of each chapter, but don't confine yourselves to solving only them. You'll write **thousands of lines of code** until you become really good – that's the life of a programmer. This book is indeed **a great start**! Seize the opportunity to come across everything of utmost importance in one place without all the wandering through the thousands of self-instruction books and articles on the Internet. Good luck!

Dr. Pavlin Dobrev *is technical director at ProSyst Labs (www.prosyst.com), a software engineer with more than 15 years' experience,* ***consultant*** *and* ***scientist****, Ph.D. in "Computer Systems, Complexes and Networks". Pavlin has made worldwide contributions in developing modern computer technologies and* ***technological standards****. He is an active member of international standardization organizations such as the OSGi Alliance (www.osgi.org) and the Java Community Process (www.jcp.org), as well as open source software initiatives such as the Eclipse Foundation (www.eclipse.org). Pavlin manages software projects and consults companies of the likes of Miele, Philips, Siemens, BMW, Bosch, Cisco Systems, France Telecom, Renault, Telefonica, Telekom Austria, Toshiba, HP, Motorola, Ford, SAP, etc. in the field of embedded applications, OSGi based automobile systems, mobile devices and home networks, integrated development environments and Java Enterprise servers for applications. He has many* ***scientific and technical publications*** *and has participated in prestigious international conferences.*

Review by Nikolay Manchev, Oracle

To become a skillful software developer, you must be ready to invest in gaining knowledge in many fields and a particular programming language is only one of them. A good developer mustn't only know the syntax and the application programming interface of the language he's chosen. He also has to possess **deep knowledge in object-oriented programming**, **data structures** and **quality code writing**. He must also back up his knowledge with serious practical experience.

When I was starting my career as a software developer more than 15 years ago, finding a comprehensive source for **learning these things was impossible**. Yes, there were books on the individual programming languages, but they only described their syntax. For the API description one had to rely on the documentation of the libraries. There were individual books devoted solely on object-oriented programming. The various algorithms and data

structures were taught at the university. There was not even a word on high-quality programming code.

Learning all these things, one piece at a time, and making the efforts to put them into a common context, was up to the one walking "the way of the programmer". Sometimes a self-taught programmer cannot manage to fill the huge gaps in their knowledge simply because they have no idea of the gaps' existence. Let me give you an example to illustrate the problem.

In the year 2000 I picked up the management of a large Java project. The team developing it consisted of 25 people and at that moment there were about 4000 classes written for the project. As a team leader, part of my job was to regularly **review the code** written by the other programmers. One day I saw how one of my colleagues had solved a standard array sorting assignment. He had written a separate, 25 lines long method implementing the trivial bubble sort algorithm. When I went to see him and asked him why he would do that instead of solving the problem with a single line of code using **`Array.Sort()`**, he started explaining how the built-in method had been "sluggish" and that it's better to write these things yourself. I told him to open the documentation and showed him that the "sluggish" method works with a **complexity** of O(n*log(n)) and his bubble sort is a prime example of bad performance with its complexity of $O(n^2)$. In the next few minutes of our conversation I made the actual discovery – my colleague had no idea what **algorithm complexity** is and his knowledge of standard algorithms was tragic. Consequently I found out he majored in an entirely different engineering discipline, not computer science. Of course, there's nothing wrong with that. His knowledge of Java was no worse than his co-workers', who had longer professional exposures than him. But that very day we noticed a gap in his professional qualification as a developer that he hadn't even suspected.

I don't want to leave you with wrong impressions from this story. Although a college student who has successfully passed his main exams in "Informatics" would definitely know the common sorting algorithms and would be able to calculate their complexity, they would also have **gaps in their education**. The sad truth is that the college education in Bulgaria in this discipline is still theoretically oriented. It has changed very little over the course of the past 15 years. Yes, programs are nowadays written in Java and C#, but these are the same programs that were written in Pascal and Ada back then.

Somewhere about a year ago I consulted a freshman student who was majoring in "Informatics" at one of Bulgaria's top state universities. When we sat down to go over his notes taken during the "Introduction to Programming" class, I was **amazed at the code his instructor had given**. The names of the methods were a mix of English and transliterated Bulgarian. There was a method **`calculate`** and a method **`rezultat`** (the Bulgarian for "result"). The variables carried the descriptive names **`a1`**, **`a2`** and **`suma`** (the Bulgarian for "sum"). Yes, there is nothing tragic in this approach, as long as it's a ten-lines-long example, but when this student takes up the job he's earned at some large project, he will be harshly rebuked by the project leader, who will have to **explain to him the coding conventions**, naming principle,

cohesion and coupling and variable life span. Then they'll find out together about the gap in his knowledge of quality code the same way my colleague and I found out about his uncertain knowledge in the field of algorithms.

Dear reader, I can boldly state that you are holding **a truly unique book** in your hands. Its contents are very carefully selected. It's well-arranged and presented with attention to details, of which only people with **tremendous practical experience** and solid scientific knowledge, like the book's chief authors Svetlin Nakov and Veselin Kolev, are capable of. Over the course of many years they have also been learning "on the fly", supplementing and expanding their knowledge. They've worked for years on huge projects, they've attended many scientific conferences and they've taught hundreds of students. They know what's **necessary for anybody striving for a career in software development** to learn and they've presented it in a manner that no other book on introduction to programming has done before. Your journey through the book's pages will lead you through the C# programming language's syntax. You'll see how to use a large part of its API. You'll learn the fundamentals of **object-oriented programming** and you'll be able to work freely with terms such as objects, events and exceptions. You'll see the most widely used **data structures** such as arrays, trees, hash tables and graphs. You'll get to know the most widely used algorithms for working with these structures and you'll come to know their pros and cons. You'll understand the concepts for creating **high-quality programming code** and you'll know what to require from your programmers when one day you become a team leader. In addition, the book will challenge you with **many practical problems** that will help you master, by the way of practice, the subject matter it covers. And if one of the problems proves too hard for you, you can always take a look at the solutions and guidelines the authors have provided.

Computer programmers make mistakes – no one is safe from that. The more capable ones make mistakes out of oversight or overwork, but the more incompetent ones – out of lack of knowledge. **Whether you become a good or a bad software developer depends entirely on you** and especially on how much you're willing to constantly invest in your knowledge – be it by attending courses, reading or practicing. But I can tell you one thing for sure – no matter how much time you invest in this book, you won't make a mistake. If some years ago someone wanting to become a software developer had asked me "Where do I start from", I wouldn't have been able to give them a definitive answer. Today I can say without worry – "**Start from this very book** (in its C# or Java version)!"

I wish you with all my heart success in mastering the secrets of C#, the .NET Framework and software development!

Nikolay Manchev *is a consultant and* ***software developer*** *with many years of experience in Java Enterprise and Service Oriented Architecture (SOA). He has worked for BEA Systems and Oracle Corporation. He's a certified developer in the programs run by Sun, BEA and Oracle. He* ***teaches software technologies*** *and* ***holds courses*** *in "Network Programming",*

*"J2EE", "Data Compression" and "High Quality Programming Code" at the Plovdiv University "Paisii Hilendarski" and at the Sofia University "St. Kliment Ohridski". He has held a number of courses for developers on Oracle technologies in Central and Eastern Europe (Hungary, Greece, Slovakia, Slovenia, Croatia and others) and has participated in international projects on incorporating J2EE based systems for security management. Works of his in the field of data compression algorithms have been accepted and presented in the USA by IEEE. Nikolay is an honorary member of the Bulgarian Association of Software Developers (BASD). He is **author of the book** "Oracle Database Security: Version 10g & 11g". You can find out more about him on his personal website: http://www.manchev.org. To contact him, use the e-mail address: nick@manchev.org.*

Review by Panayot Dobrikov, SAP AG

The book at hand is an **incredibly good introduction to programming for beginners** and is a primary example of the notion (promoted by Wikipedia and others) to create and distribute easy to understand knowledge that is not only ***free of charge***, but is of **incredibly high quality** as well.

***Panayot Dobrikov** is program director at SAP AG and co-author of the book "Programming = ++Algorithms;". You can find out more about him on his personal website: http://indyana.hit.bg.*

Review by Lyubomir Ivanov, Mobiltel

If someone had told me 5 or 10 years ago that there would be a book from which to learn the basics of managing people and projects – budgeting, finances, psychology, planning, etc., I wouldn't have believed them. I wouldn't even believe them at this very moment. For each of these topics there are tens of books that must be read.

If someone had told me that there would be a book from which we can **learn the fundamentals of programming** essential to every software developer – I still wouldn't have believed them.

I remember my time as a novice programmer and a college student – I was reading several books on programming languages, several others on algorithms and data structures, and a third set of books on writing high-quality code. Very few of them helped me **to think algorithmically** and to work out an **approach for solving the everyday problems** I came across in my practice. None of them gave me an overview of everything I had to know as a computer programmer and a software engineer. The only things that helped me were being stubborn and reinventing the wheel.

Today I read this book and I'm happy that finally, although a bit too late for me, someone got down to writing **The Book that will help any beginner programmer solve the puzzle of programming** – a modern programming language, data structures, quality code, algorithmic thinking and problem solving. This is the book that you should take up programming from, if you

want to master the art of quality programming. Whether you choose the Java or C# version of this book, it doesn’t really matter. What matters is that you must **learn to think as a programmer** and solve the problems you encounter when writing software; the programming language is just a tool you can change for another at any given time.

This book **isn’t only for beginners**. Even programmers with many years of experience can learn something from it. I recommend it to every software developer who would like to realize what they didn’t know up until now.

Have a nice time reading!

Lyubomir Ivanov *is the* ***manager*** *of the "Data and Mobile Applications" department at Mobiltel EAD (part of Mobilkom Austria) where he engages in developing and integrating IT solutions for the telecommunications industry.*

Review by Hristo Deshev, Entrepreneur

It’s surprising what a large percentage of programmers don’t pay attention to the little things like **variable names and good code structure**. These things pile up and, in the end, make the difference between a well-written piece of software and a bowl of spaghetti. This book teaches **discipline and "hygiene" in code writing** along with the very basic concepts in programming and that will undoubtedly make you a **professional**.

Hristo Deshev*, software craftsman*

Review by Hristo Radkov, Clever IT (London, UK)

Fantastic book! It gives the start to any developer geek who wants to develop into a software prodigy. While you can learn from the quick learning books for dummies to do coding that “just works” and this is the level expected in many of the small software development houses around, you can never leave a trace in the software world without understanding the **fundamental concepts of programming**. Yes, you can still develop software applications and use the goodies of the .NET framework, but just **use and not create or innovate**.

If you’d like to ever achieve **architectural excellence** and be able to confidently and proudly say you have developed a good piece of software that will stay there and serve its purpose for years, you need to understand just how the technologies you use in everyday live (e.g. ASP.NET, MVC, WPF, WCF, LINQ, Sockets, Task Parallel Library) work, but how they have been developed and optimized to become what they are. Only then would you save precious time in finding how to do things efficiently with these technologies, because that knowledge will naturally come from **what you have learned from this book**. And the same applies to understanding the widely recommended in the world of programming nowadays design patterns, architectures and techniques.

The book will allow you to **prepare yourself to think, design and program** optimally as a concept and mindset with any object oriented language you might ever use not just C# or .NET Framework.

Many banking systems here in London have a main requirement to be "real-time" data servers to thousands of users with minimum delays and interruptions, and this book provides the basics which if you lack you cannot work on such systems successfully, ever.

This **fundamental knowledge** distinguishes the excellent and accomplished developer, whose code would rarely require optimizations and would therefore save direct and indirect costs to their employer from the general developers who unfortunately are the prevailing part of the programmers you would meet in your career. The accomplished specialists evolve and progress into senior positions much easier when having the technical arguments and the mentality to be creative and visionary, avoiding the difficulties of technology gap limitations the mass around you have.

So, **read the book carefully and diligently** to become one!

Hristo Radkov *is a Chief software architect and* ***Co-founder at Clever IT****, a software services, best coding practices and architecture consulting company based in London, United Kingdom. With over 15 years of experience as a Developer, Team leader, Development manager, Head of IT and Software Architect he has done projects professionally with C++, Java and C#, eventually remaining completely on the side of the Microsoft Technologies after the very first release of* ***.NET Framework****, becoming recognized by the industry Microsoft Technology Software Development Best Practices and Cloud Programming Expert, with MCPD, MCSD.NET, MCDBA and MCTS awards. Hristo is* ***co-author of the books*** *"Programming for the .NET Framework (vol. 1 & 2)" and has been* ***instructor*** *for .NET and Design Patterns for many years. His company Clever IT is consulting top financial institutions and FTSE 100 corporations with multibillion valuations on the World Stock Exchanges. You can find more about him on www.radkov.com or linked-in at Hristo Radkov. To contact him, use the e-mail address: hradkov@clevit.com.*

License

The book and all its study materials are distributed freely under the following **license**:

Common Definitions

1. The present license defines the terms and conditions for using and distributing the "**study materials**" and the **book "Fundamentals of Computer Programming with C#"**, developed by a team of volunteers under the guidance of Svetlin Nakov (www.nakov.com).
2. The **study materials** consist of:

- the book (textbook) on "Fundamentals of Computer Programming with C#"
- sample source code
- demo programs
- exercise problems
- presentation slides
- video materials

3. The study materials are available for **free download** according to the terms and conditions specified in this license at the official website of the project: www.introprogramming.info.
4. **Authors** of the study materials are the persons who participated in their creation.
5. **User** of the study materials is anybody who uses or accesses these materials or portions of them.

Rights and Limitations of the Users

1. Users **may**:
   - distribute **free of charge** unaltered copies of the study materials in electronic or paper format;
   - use the study materials or portions of them, including the examples, demos, exercises and presentation slides or their modifications, for all intents and purposes, including **educational and commercial projects**, provided they clearly **specify the original source**, the original author(s) of the corresponding text or source code, this license and the website www.introprogramming.info;
   - distribute **free of charge** portions of the study materials or modified copies of them (including translating them into other languages or adapting them to other programming languages and platforms), but only by **explicitly mentioning the original source** and the authors of the corresponding text, source code or other material, this license and the official website of the project: www.introprogramming.info.
2. Users **may not**:
   - **distribute for profit** the study materials or portions of them, with the exception of the source code;
   - **remove this license** from the study materials when modifying them for own needs.

Rights and Limitations of the Authors

1. Every author has non-exclusive rights on the products of his / her own work contributing to build the study materials.
2. The authors have the right to use the products of their contribution for any purpose, including modifying them and distributing them for profit.
3. The rights on all study materials written in joint authorship belong to all co-authors together.
4. The authors may not distribute for profit study materials they've written in joint authorship without the explicit permission of all other co-authors.

Resources Coming with the Book

This book "Fundamentals of Computer Programming with C#" comes with a rich set of resources: official **web site**, official discussion **forum**, presentation **slides** for each chapter of the book, **video lessons** for each chapter of the book and **Facebook fan page**.

The Book's Website

The **official website** of the book "Introduction to programming with C#" is available at: **www.introprogramming.info**. At book's web site you can freely **download** the book and many related resources:

- The whole **book** in several electronic formats (PDF / DOC / DOCX / HTML / Kindle / etc.)
- The source code of the **examples** (demos) for each chapter
- **Video lessons** covering the entire book content with live demos and detailed explanations (in English and in Bulgarian)
- PowerPoint **presentations slides** for each chapter, ready for instructors who want to teach programming (in English)
- The **exercises** and **solutions guidelines** for each chapter
- **Solutions to all problems** from the book + explanation of the algorithm and the source code for each solution + tests (in Bulgarian)
- Interactive **Mind maps** for each book chapter
- The book in **Bulgarian** language (the original)
- A **Java version** of the book (with all content and examples adapter to Java programming language).

Discussion Forum

The discussion forum where you can find solutions to almost all problems from the book is available at: **forums.academy.telerik.com**.

This forum is created for discussions among the participants in Telerik Software Academy's courses who go through this book during the first few months of their training and mandatorily **solve all problems** in the exercise sections. Most people "living" in the forum are Bulgarian but everyone speaks English so you are invited to ask your questions about the book exercises in English.

In the forum you'll find **comments and solutions** submitted by students and readers of the book, as well as by the trainers at the Software Academy. Just search thoroughly enough and you'll find several solutions to all problems in the book (with no exceptions). Every year **thousands of participants** in Telerik Software Academy solve problems from this book and share their solutions and the difficulties they've encountered, so simply search thoroughly in the forum or ask, if you can't get to a solution for a particular problem.

Presentation Slides Coming with the Book

This book is used in many universities, colleges, schools and organizations as a textbook on computer programming, C#, data structures and algorithms. To help instructors teach the lessons following this book we have prepared **PowerPoint presentation slides** for each chapter of the book. Instructors are welcome to use the slides **free of charge** under the license agreement stated above. The authors' team will be happy to find out that this book and its study materials and presentation slides are helping people all over the world to learn programming. This is the primary goal of the project: **to teach computer programming fundamentals**, in complete, simple, structured, understandable way, free of charge. You may find the PowerPoint slides in English at the book's official web site: **www.introprogramming.info**.

Video Materials for Self-Education with the Book

As part of the Telerik Software Academy program (academy.telerik.com) and, in particular, the free course "Fundamentals of C# Programming", **videos of all lectures** on the subject matter in this book have been recorded. The video materials in English and Bulgarian can be found at C# book's official web site: **introprogramming.info**.

If you speak Bulgarian you might be interested in Telerik Software Academy's video channel in YouTube: **youtube.com/TelerikAcademy**. It provides for free **thousands video lessons** on programming and software development.

Interactive Mind Maps

As part of the book we created a set of interactive **mind maps** to visualize its content and to improve the level of memorization. We have a few mind maps for each chapter that visually illustrates its content and a global mind map of the entire book. The mind maps are available at the book's web site: http://www.introprogramming.info/english-intro-csharp-book/mind-maps/.

Mind Maps on the Book "Fundamentals of Computer Programming with C#"

C# Book Fan Club

For the fans of the book "Introduction to Programming with C#" we have a **Facebook page**: **www.facebook.com/IntroCSharpBook**.

Svetlin Nakov, PhD,

Manager of the "Technical Training" Department,

Telerik Software Academy, Telerik Corporation,

August 24th, 2013

Chapter 1. Introduction to Programming

In This Chapter

In this chapter we will take a look at the **basic programming terminology** and we will **write our first C# program**. We will familiarize ourselves with programming – what it means and its connection to computers and programming languages.

Briefly, we will review the different **stages of software development**.

We will introduce the C# language, the .NET platform and the different Microsoft technologies used in software development. We will examine what tools we need **to program in C#**. We will use the C# language to write our first computer program, compile and run it from the command line as well as from Microsoft **Visual Studio** integrated development environment. We will review the MSDN Library – the documentation of the .NET Framework. It will help us with our exploration of the features of the platform and the language.

What Does It Mean "To Program"?

Nowadays computers have become irreplaceable. We use them to solve complex problems at the workplace, look for driving directions, have fun and communicate. They have countless applications in the business world, the entertainment industry, telecommunications and finance. It's not an over-statement to say that computers build the neural system of our contemporary society and it is difficult to imagine its existence without them.

Despite the fact that computers are so wide-spread, **few people know how they really work**. In reality, it is not the computers, but the programs (the software), which run on them, that matter. It is the **software** that makes computers valuable to the end-user, allowing for many different types of services that change our lives.

How Do Computers Process Information?

In order to understand what it means to program, we can roughly compare a computer and its operating system to a large factory with all its workshops, warehouses and transportation. This rough comparison makes it easier to imagine the level of complexity present in a contemporary computer. There are many processes running on a computer, and they represent the workshops and production lines in a **factory**. The hard drive, along with the

files on it, and the operating memory (RAM) represent the warehouses, and the different protocols are the transportation systems, which provide the input and output of information.

The different types of products made in a factory come from different workshops. They use raw materials from the warehouses and store the completed goods back in them. The raw materials are transported to the warehouses by the suppliers and the completed product is transported from the warehouses to the outlets. To accomplish this, different types of transportation are used. Raw materials enter the factory, go through different stages of processing and leave the factory transformed into products. Each factory converts the raw materials into a product ready for consumption.

The computer is a machine for information processing. Unlike the factory in our comparison, for the computer, the raw material and the product are the same thing – information. In most cases, the input information is taken from any of the warehouses (files or RAM), to where it has been previously transported. Afterwards, it is processed by one or more processes and it comes out modified as a new product. Web based applications serve as a prime example. They use HTTP to transfer raw materials and products, and information processing usually has to do with extracting content from a database and preparing it for visualization in the form of HTML.

Managing the Computer

The whole process of manufacturing products in a factory has many levels of management. The separate machines and assembly lines have operators, the workshops have managers and the factory as a whole is run by general executives. Every one of them controls processes on a different level. The machine operators serve on the lowest level – they control the machines with buttons and levers. The next level is reserved for the workshop managers. And on the highest level, the general executives manage the different aspects of the manufacturing processes in the factory. They do that by issuing orders.

It is the same with computers and software – they have many levels of management and control. The lowest level is managed by the **processor** and its registries (this is accomplished by using machine programs at a low level) – we can compare it to controlling the machines in the workshops. The different responsibilities of the **operating system** (Windows 7 for example), like the file system, peripheral devices, users and communication protocols, are controlled at a higher level – we can compare it to the management of the different workshops and departments in the factory. At the highest level, we can find the **application software**. It runs a whole ensemble of processes, which require a huge amount of processor operations. This is the level of the general executives, who run the whole factory in order to maximize the utilization of the resources and to produce quality results.

The Essence of Programming

The essence of programming is to control the work of the computer on all levels. This is done with the help of "orders" and "commands" from the programmer, also known as programming instructions. To "program" means **to organize the work of the computer through sequences of instructions**. These commands (instructions) are given in written form and are implicitly followed by the computer (respectively by the operating system, the CPU and the peripheral devices).

To “program” means writing sequences of instructions in order to organize the work of the computer to perform something. These sequences of instructions are called “computer programs” or “scripts”.

A sequence of steps to achieve, complete some work or obtain some result is called an **algorithm**. This is how **programming is related to algorithms**. Programming involves describing what you want the computer to do by a sequence of steps, by **algorithms**.

Programmers are the people who create these instructions, which control computers. These instructions are called **programs**. Numerous programs exist, and they are created using different kinds of **programming languages**. Each language is oriented towards controlling the computer on a different level. There are languages oriented towards the machine level (the lowest) – **Assembler** for example. Others are most useful at the system level (interacting with the operating system), like **C**. There are also high level languages used to create application programs. Such languages include **C#**, Java, C++, PHP, Visual Basic, Python, Ruby, Perl, JavaScript and others.

In this book we will take a look at **the C# programming language** – a modern high level language. When a programmer uses C#, he gives commands in high level, like from the position of a general executive in a factory. The **instructions** given in the form of programs written in C# can access and control almost all computer resources directly or via the operating system. Before we learn how to write simple C# programs, let’s take a good look at the different stages of software development, because programming, despite being the most important stage, is not the only one.

Stages in Software Development

Writing software can be a very complex and time-consuming task, involving a whole team of software engineers and other specialists. As a result, many methods and practices, which make the life of programmers easier, have emerged. All they have in common is that the development of each software product goes through several different **stages**:

- Gathering the **requirements** for the product and creating a task;
- **Planning** and preparing the architecture and design;

- **Implementation** (includes the writing of program code);
- Product trials (**testing**);
- **Deployment** and exploitation;
- **Support**.

Implementation, testing, deployment and support are mostly accomplished using programming.

Gathering the Requirements

In the beginning, only the idea for a certain product exists. It includes a list of **requirements**, which define actions by the user and the computer. In the general case, these actions make already existing activities easier – calculating salaries, calculating ballistic trajectories or searching for the shortest route on Google maps are some examples. In many cases the software implements a previously nonexistent functionality such as automation of a certain activity.

The **requirements** for the product are usually defined in the form of documentation, written in English or any other language. There is no programming done at this stage. The requirements are defined by experts, who are familiar with the problems in a certain field. They can also write them up in such a way that they are easy to understand by the programmers. In the general case, these experts are not programming specialists, and they are called **business analysts**.

Planning and Preparing the Architecture and Design

After all the requirements have been gathered comes **the planning stage**. At this stage, a technical plan for the implementation of the project is created, describing the platforms, technologies and the initial architecture (design) of the program. This step includes a fair amount of creative work, which is done by software engineers with a lot of experience. They are sometimes called **software architects**. According to the requirements, the following parts are chosen:

- The **type of the application** – for example console application, desktop application (GUI, Graphical User Interface application), client-server application, Web application, Rich Internet Application (RIA), mobile application, peer-to-peer application or other;
- The **architecture** of the software – for example single layer, double layer, triple layer, multi-layer or SOA architecture;
- The **programming language** most suitable for the implementation – for example C#, Java, PHP, Python, Ruby, JavaScript or C++, or a combination of different languages;
- The **technologies** that will be used: platform (Microsoft .NET, Java EE, LAMP or another), database server (Oracle, SQL Server, MySQL, NoSQL

database or another), technologies for the user interface (Flash, JavaServer Faces, Eclipse RCP, ASP.NET, Windows Forms, Silverlight, WPF or another), technologies for data access (for example Hibernate, JPA or ADO.NET Entity Framework), reporting technologies (SQL Server Reporting Services, Jasper Reports or another) and many other combinations of technologies that will be used for the implementation of the various parts of the software system.

- The **development frameworks** that will simplify the development, e.g. ASP.NET MVC (for .NET), Knockout.js (for JavaScript), Rails (for Ruby), Django (for Python) and many others.
- The number and skills of the **people** who will be part of the development team (big and serious projects are done by large and experienced teams of developers);
- The **development plan** – separating the functionality in stages, resources and deadlines for each stage.
- Others (size of the team, locality of the team, methods of communication etc.).

Although there are many rules facilitating the correct analysis and planning, a fair amount of intuition and insight is required at this stage. This step predetermines the further advancement of the development process. There is no programming done at this stage, only preparation.

Implementation

The stage, most closely connected with programming, is the implementation stage. At this phase, the program (application) is implemented (written) according to the given task, design and architecture. **Programmers** participate by **writing the program** (source) code. The other stages can either be short or completely skipped when creating a small project, but the implementation always presents; otherwise the process is not software development. This book is dedicated mainly to describing the skills used during implementation – creating a **programmer's mindset** and building the knowledge to use all the resources provided by the C# language and the .NET platform, in order to create software applications.

Product Testing

Product testing is a very important stage of software development. Its purpose is to make sure that all the requirements are strictly followed and covered. This process can be implemented manually, but the preferred way to do it is by **automated tests**. These tests are small programs, which automate the trials as much as possible. There are parts of the functionality that are very hard to automate, which is why product trials include automated as well as manual procedures to ensure the quality of the code.

The testing (trials) process is implemented by **quality assurance engineers (QAs)**. They work closely with the programmers to find and correct errors (bugs) in the software. At this stage, it is a priority to find defects in the code and almost no new code is written.

Many **defects** and **errors** are usually found during the testing stage and the program is sent back to the implantation stage. These two stages are very closely tied and it is common for a software product to switch between them many times before it covers all the requirements and is ready for the deployment and usage stages.

Deployment and Operation

Deployment is the process which **puts a given software product into exploitation**. If the product is complex and serves many people, this process can be the slowest and most expensive one. For smaller programs this is a relatively quick and painless process. In the most common case, a special program, called installer, is developed. It ensures the quick and easy installation of the product. If the product is to be deployed at a large corporation with tens of thousands of copies, additional supporting software is developed just for the deployment. After the **deployment** is successfully completed, the product is ready for **operation**. The next step is to train employees to use it.

An example would be the deployment of a new version of Microsoft Windows in the state administration. This includes **installation** and **configuration** of the software as well as **training** employees how to use it.

The deployment is usually done by the team who has worked on the software or by trained **deployment specialists**. They can be system administrators, database administrators (DBA), system engineers, specialized consultants and others. At this stage, almost no new code is written but the existing code is tweaked and configured until it covers all the specific requirements for a successful deployment.

Technical Support

During the exploitation process, it is inevitable that **problems will appear**. They may be caused by many factors – errors in the software, incorrect usage or faulty configuration, but most problems occur when the users change their requirements. As a result of these problems, the software loses its abilities to solve the business task it was created for. This requires additional involvement by the developers and the **support experts**. The support process usually continues throughout the whole life-cycle of the software product, regardless of how good it is.

The support is carried out by the development team and by specially trained **support experts**. Depending on the changes made, many different people may be involved in the process – business analysts, architects, programmers, QA engineers, administrators and others.

For example, if we take a look at a software program that calculates salaries, it will need to be **updated** every time the tax legislation, which concerns the serviced accounting process, is changed. The support team's intervention will be needed if, for example, the hardware of the end user is changed because the software will have to be installed and configured again.

Documentation

The documentation stage is not a separate stage but accompanies all the other stages. **Documentation** is an important part of software development and aims to pass knowledge between the different participants in the development and support of a software product. Information is passed along between different stages as well as within a single stage. The **development documentation** is usually created by the developers (architects, programmers, QA engineers and others) and represents a combination of documents.

Software Development Is More than Just Coding

As we saw, software development is much more than just coding (writing code), and it includes a number of other processes such as: requirements analysis, design, planning, testing and support, which require a wide variety of specialists called **software engineers**. Programming is just a small, but very essential part of software development.

In this book we will focus solely on programming, because it is the only process, of the above, without which, we cannot develop software.

Our First C# Program

Before we continue with an in depth description of the C# language and the .NET platform, let's take a look at a simple example, illustrating how a **program written in C#** looks like:

```
class HelloCSharp
{
  static void Main(string[] args)
  {
    System.Console.WriteLine("Hello C#!");
  }
}
```

The only thing this program does is to **print the message "Hello, C#!"** on the default output. It is still early to execute it, which is why we will only take a look at its structure. Later we will describe in full how to compile and run a given program from the command prompt as well as from a development environment.

How Does Our First C# Program Work?

Our first program consists of three logical parts:

- Definition of a class **HelloCSharp**;
- Definition of a method **Main()**;
- Contents of the method **Main()**.

Defining a Class

On the first line of our program we define a class called **HelloCSharp**. The simplest definition of a class consists of the keyword **class**, followed by its name. In our case the name of the class is **HelloCSharp**. The content of the class is located in a block of program lines, surrounded by curly brackets: **{}**.

Defining the Main() Method

On the third line we define a method with the name **Main()**, which is the starting point for our program. Every program written in C# starts from a **Main()** method with the following title (signature):

```
static void Main(string[] args)
```

The method must be declared as shown above, it must be **static** and **void**, it must have a name **Main** and as a list of parameters it must have only one parameter of type **array of string**. In our example the parameter is called **args** but that is not mandatory. This parameter is not used in most cases so it can be omitted (it is optional). In that case the entry point of the program can be **simplified** and will look like this:

```
static void Main()
```

If any of the aforementioned requirements is not met, the program will compile but it will not start because the starting point is not defined correctly.

Contents of the Main() Method

The content of every method is found after its signature, surrounded by opening and closing curly brackets. On the next line of our sample program we use the system object **System.Console** and its method **WriteLine()** to print a message on the default output (the console), in this case "Hello, C#!".

In the **Main()** method we can write a random sequence of expressions and they will be executed in the order we assigned to them.

More information about expressions can be found in chapter "Operators and Expressions", working with the console is described in chapter "Console Input and Output", classes and methods can be found in chapter "Defining Classes".

C# Distinguishes between Uppercase and Lowercase!

The C# language distinguishes between **uppercase** and **lowercase** letters so we should use the correct casing when we write C# code. In the example above we used some keywords like **class**, **static**, **void** and the names of some of the system classes and objects, such as **System.Console**.

Be careful when writing! The same thing, written in upper-case, lower-case or a mix of both, means different things in C#. Writing `Class` **is different from** `class` **and** `System.Console` **is different from** `SYSTEM.CONSOLE`**.**

This rule applies to all elements of your program: keywords, names of variables, class names etc.

The Program Code Must Be Correctly Formatted

Formatting is adding characters such as spaces, tabs and new lines, which are insignificant to the compiler and they give the code a **logical structure** and make it **easier to read**. Let's for example take a look at our first program (the short version of the **Main()** method):

```
class HelloCSharp
{
   static void Main()
   {
      System.Console.WriteLine("Hello C#!");
   }
}
```

The program contains seven lines of code and some of them are indented more than others. All of that can be **written without tabs** as well, like so:

```
class HelloCSharp
{
static void Main()
{
System.Console.WriteLine("Hello C#!");
}
}
```

Or on the same line:

```
class HelloCSharp{static void Main(){System.Console.WriteLine(
"Hello C#!");}}
```

Or even like this:

```
                                                          class
    HelloCSharp
{
    static void                                          Main()
    {                                          System          .
Console.WriteLine("Hello C#!")          ;}          }
```

The examples above will compile and run exactly like the formatted code but they are more **difficult to read and understand**, and therefore difficult to modify and maintain.

Never let your programs contain unformatted code! That severely reduces program readability and leads to difficulties for later modifications of the code.

Main Formatting Rules

If we want our code to be correctly formatted, we must follow several important **rules regarding indentation**:

- Methods are **indented** inside the definition of the class (move to the right by one or more **[Tab]** characters);
- Method contents are indented inside the definition of the method;
- The opening curly bracket **{** must be on its own line and placed exactly under the method or class it refers to;
- The closing curly bracket **}** must be on its own line, placed exactly vertically under the respective opening bracket (with the same indentation);
- All class names must start with a capital letter;
- Variable names must begin with a lower-case letter;
- Method names must start with a capital letter;

Code indentation follows a very simple rule: when some piece of code is logically inside another piece of code, it is indented (moved) on the right with a single [Tab]. For example if a method is defined inside a class, it is indented (moved to the right). In the same way if a method body is inside a method, it is indented. To simplify this, we can assume that when we have the character "**{**", all the code after it until its closing "**}**" should be indented on the right.

File Names Correspond to Class Names

Every C# program consists of **one or several class definitions**. It is accepted that each class is defined in a separate file with a name corresponding to the class name and a **.cs** extension. When these requirements are not met, the program will still work but navigating the code

will be difficult. In our example, the class is named **HelloCSharp**, and as a result we must save its source code in a file called **HelloCSharp.cs**.

The C# Language and the .NET Platform

The first version of **C#** was developed by Microsoft between 1999 and 2002 and was officially released to the public in 2002 as a part of the .NET platform. **The .NET platform** aims to make software development for Windows easier by providing a new quality approach to programming, based on the concepts of the "**virtual machine**" and "**managed code**". At that time the Java language and platform reaped an enormous success in all fields of software development; C# and .NET were Microsoft's natural response to the Java technology.

The C# Language

C# is a modern, general-purpose, object-oriented, high-level programming language. Its syntax is similar to that of C and C++ but many features of those languages are not supported in C# in order to simplify the language, which makes programming easier.

The C# programs consist of **one or several files** with a **.cs** extension, which contain definitions of classes and other types. These files are compiled by the C# compiler (**csc**) to executable code and as a result assemblies are created, which are files with the same name but with a different extension (**.exe** or **.dll**). For example, if we compile **HelloCSharp.cs**, we will get a file with the name **HelloCSharp.exe** (some additional files will be created as well, but we will not discuss them at the moment).

We can run the compiled code like any other program on our computer (by double clicking it). If we try to execute the compiled C# code (for example **HelloCSharp.exe**) on a computer that does not have the .NET Framework, we will receive an error message.

Keywords

C# uses the following **keywords** to build its programming constructs (the list is taken from MSDN in March 2013 and may not be complete):

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null

object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while	

Since the creation of the first version of the C# language, **not all keywords are in use**. Some of them were added in later versions. The main program elements in C# (which are defined and used with the help of keywords) are **classes**, **methods**, **operators**, **expressions**, **conditional statements**, **loops**, **data types**, **exceptions** and few others. In the next few chapters of this book, we will review in details all these programming constructs along with the use of the most of the keywords from the table above.

Automatic Memory Management

One of the biggest advantages of the .NET Framework is the **built-in automatic memory management**. It protects the programmers from the complex task of manually allocating memory for objects and then waiting for a suitable moment to release it. This significantly increases the developer productivity and the quality of the programs written in C#.

In the .NET Framework, there is a special component of the CLR that looks after memory management. It is called a "**garbage collector**" (automated memory cleaning system). The garbage collector has the following main tasks: to check when the allocated memory for variables is no longer in use, to release it and make it available for allocation of new objects.

It is important to note that it is not exactly clear at what moment the memory gets cleaned of unused objects (local variables for example). According to the C# language specifications, it happens at some moment after a given variable gets out of scope but it is not specified, whether this happens instantly, after some time or when the available memory becomes insufficient for the normal program operation.

Independence from the Environment and the Programming Language

One of the advantages of .NET is that programmers using different **.NET languages** can easily exchange their code. For example a **C#** programmer can use the code written by another programmer in **VB.NET**, **Managed C++** or **F#**. This is possible because the programs written in different .NET

languages share a common system of data types, execution infrastructure and a unified format of the compiled code (**assemblies**).

A big advantage of the .NET technology is the ability to run code, which is written and compiled only once, on **different operating systems** and hardware devices. We can compile a C# program in a Windows environment and then execute it under Windows, Windows Mobile, Windows RT or Linux. Officially Microsoft only supports the .NET Framework on Windows, Windows Mobile and Windows Phone, but there are third party vendors that offer .NET implementation on other operating systems.

Mono (.NET for Linux)

One example of .NET implementation for non-Windows environment is the **open-source project Mono** (www.mono-project.com). It implements the .NET Framework and most of its accompanying libraries for Linux, FreeBSD, iPhone and Android. Mono is **unofficial .NET implementation** and some features may work not exactly as expected. It does implement well the core .NET standards (such as C# compiler and CLR) but does not support fully the latest .NET technologies and framework like WPF and ASP.NET MVC.

Microsoft Intermediate Language (MSIL)

The idea for independence from the environment has been set in the earliest stages of creation of the .NET platform and is implemented with the help of a little trick. The output code is not compiled to instructions for a specific microprocessor and does not use the features of a specific operating system; it is compiled to the so called **Microsoft Intermediate Language (MSIL)**. This **MSIL** is not directly executed by the microprocessor but from a virtual environment called **Common Language Runtime (CLR)**.

Common Language Runtime (CLR) – the Heart of .NET

In the very center of the .NET platform beats its heart – the **Common Language Runtime (CLR)** – the environment that controls the execution of the managed code (**MSIL** code). It ensures the execution of .NET programs on different hardware platforms and operating systems.

CLR is an abstract computing machine (**virtual machine**). Similarly to physical computers, it supports a set of instructions, registries, memory access and input-output operations. CLR ensures a **controlled execution** of the .NET programs using the full capabilities of the processor and the operating system. CLR also carries out the **managed access** to the memory and the other resources of the computer, while adhering to the access rules set when the program is executed.

The .NET Platform

The .NET platform contains the **C# language**, **CLR** and many auxiliary instruments and **libraries** ready for use. There are a few versions of .NET according to the targeted user group:

- **.NET Framework** is the most common version of the .NET environment because of its general purpose. It is used in the development of console applications, Windows applications with a graphical user interface, web applications and many more.
- **.NET Compact Framework** (CF) is a "light" version of the standard .NET Framework and is used in the development of applications for mobile phones and other PDA devices using Windows Mobile Edition.
- **Silverlight** is also a "light" version of the .NET Framework, intended to be executed on web browsers in order to implement multimedia and Rich Internet Applications.
- **.NET for Windows Store apps** is a subset of .NET Framework designed for development and execution of .NET applications in **Windows 8** and **Windows RT** environment (the so called Windows Store Apps).

.NET Framework

The standard version of **the .NET platform** is intended for development and use of console applications, desktop applications, Web applications, Web services, Rich Internet Applications, mobile applications for tablets and smart phones and many more. Almost all .NET developers use the standard version.

.NET Technologies

Although the **.NET platform is big and comprehensive**, it does not provide all the tools required to solve every problem in software development. There are many independent software developers, who expand and add to the standard functionality offered by the .NET Framework. For example, companies like the Bulgarian software corporation **Telerik** develop subsidiary sets of **components**. These components are used to create graphical user interfaces, Web content management systems, to prepare reports and they make application development easier.

The .NET Framework extensions are software components, which can be reused when developing .NET programs. Reusing code significantly facilitates and simplifies software development, because it provides solutions for common problems, offers implementations of complex algorithms and technology standards. The contemporary programmer uses **libraries and components** every day, and saves a lot of effort by doing so.

Let's look at the following example – software that **visualizes data** in the form of charts and diagrams. We can use a **library**, written in .NET, which draws the charts. All that we need to do is input the correct data and the

library will draw the charts for us. It is very convenient and efficient. Also it leads to reduction in the production costs because the programmers will not need to spend time working on additional functionality (in our case drawing the charts, which involves complex mathematical calculations and controlling the graphics card). The application itself will be of higher quality because the extension it uses is developed and supported by specialists with more experience in that specific field.

Software technologies are sets of classes, modules, libraries, programming models, tools, patterns and best practices addressing some **specific problem** in software development. There are general software technologies, such as Web technologies, mobile technologies, technologies for computer graphics and technologies related to some platform such as .NET or Java.

There are many **.NET technologies** serving for different areas of .NET development. Typical examples are the Web technologies (like **ASP.NET** and **ASP.NET MVC**), allowing fast and easy creation of dynamic Web applications and .NET mobile technologies (like **WinJS**), which make possible the creation of rich user interface multimedia applications working on the Internet.

.NET Framework by default includes as part of itself many technologies and **class libraries** with standard functionality, which developers can use. For example, there are ready-to-use classes in the **system library** working with mathematical functions, calculating logarithms and trigonometric functions (**`System.Math`** class). Another example is the library dealing with networks (**`System.Net`**), it has a built-in functionality to send e-mails (using the **`System.Net.Mail.MailMessage`** class) and to download files from the Internet (using **`System.Net.WebClient`**).

A **.NET technology** is the collection of .NET classes, libraries, tools, standards and other programming means and established development models, which determine the technological framework for creating a certain type of application. A **.NET library** is a collection of .NET classes, which offer certain ready-to-use functionality. For example, **ADO.NET** is a technology offering standardized approach to accessing relational databases (like Microsoft SQL Server and MySQL). The classes in the package (namespace) **`System.Data.SqlClient`** are an example of **.NET library**, which provide functionality to connect an SQL Server through the ADO.NET technology.

Some of the technologies developed by software developers outside of Microsoft become wide-spread and as a result establish themselves as technology standards. Some of them are noticed by Microsoft and later are added to the next iteration of the .NET Framework. That way, the .NET platform is constantly evolving and expanding with new **libraries and technologies**. For instance, the object-relational mapping technologies initially were developed as independent projects and products (like the open code project NHibernate and Telerik's OpenAccess ORM). After they gained enormous popularity, their inclusion in the .NET Framework became a necessity. And this is how the LINQ-to-SQL and ADO.NET Entity Framework technologies were born, respectively in .NET 3.5 and .NET 4.0.

Application Programming Interface (API)

Each .NET library or technology is utilized by creating objects and calling their methods. The set of public classes and methods in the programming libraries is called **Application Programming Interface** or just **API**. As an example we can look at the .NET API itself; it is a set of .NET class libraries, expanding the capabilities of the language and adding high-level functionality. All .NET technologies offer a **public API**. The technologies are often referred to simply as API, which adds certain functionality. For example: API for working with files, API for working with charts, API for working with printers, API for reading and creating Word and Excel documents, API for creating PDF documents, Web development API, etc.

.NET Documentation

Very often it is necessary to document an API, because it contains many namespaces and classes. Classes contain methods and parameters. Their purpose is not always obvious and **needs to be explained**. There are also inner dependencies between the separate classes, which need to be explained in order to be used correctly. These explanations and technical instructions on how to use a given technology, library or API, are called **documentation**. The documentation consists of a collection of documents with technical content.

The .NET Framework also has a **documentation** officially developed and supported by Microsoft. It is publicly available on the Internet and is also distributed with the .NET platform as a collection of documents and tools for browsing and searching.

The **MSDN Library** is Microsoft's official documentation for all their products for developers and software technologies. The .NET Framework's technical documentation is part of the MSDN Library and can be found here: http://msdn.microsoft.com/en-us/library/vstudio/gg145045.aspx. The above screenshot shows how it might look like (for .NET version 4.5).

What We Need to Program in C#?

After we made ourselves familiar with the **.NET platform**, **.NET libraries** and **.NET technologies**, we can move on to writing, compiling and executing C# programs.

In order to program in C#, we need two basic things – an installed **.NET Framework** and a **text editor**. We need the text editor to write and edit the C# code and the .NET Framework to compile and execute it.

.NET Framework

By default, the **.NET Framework** is installed along with Windows, but in old Windows versions it could be missing. To install the .NET Framework, we must download it from Microsoft's website (http://download.microsoft.com). It is best if we download and install the latest version.

Do not forget that we need to install the .NET Framework before we begin! Otherwise, we will not be able to compile and execute the program.

If we run Windows 8 or Windows 7, the .NET Framework will be already installed as part of Windows.

Text Editor

The **text editor** is used to write the **source code** of the program and to save it in a file. After that, the code is compiled and executed. There are many text editing programs. We can use Windows' built-in Notepad (it is very basic and inconvenient) or a better free text editor like Notepad++ (notepad-plus.sourceforge.net) or PSPad (www.pspad.com).

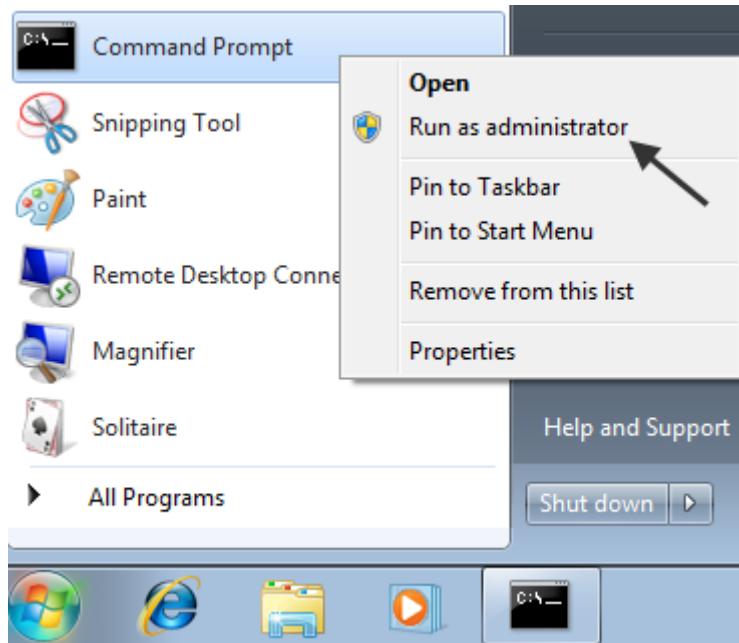
Compilation and Execution of C# Programs

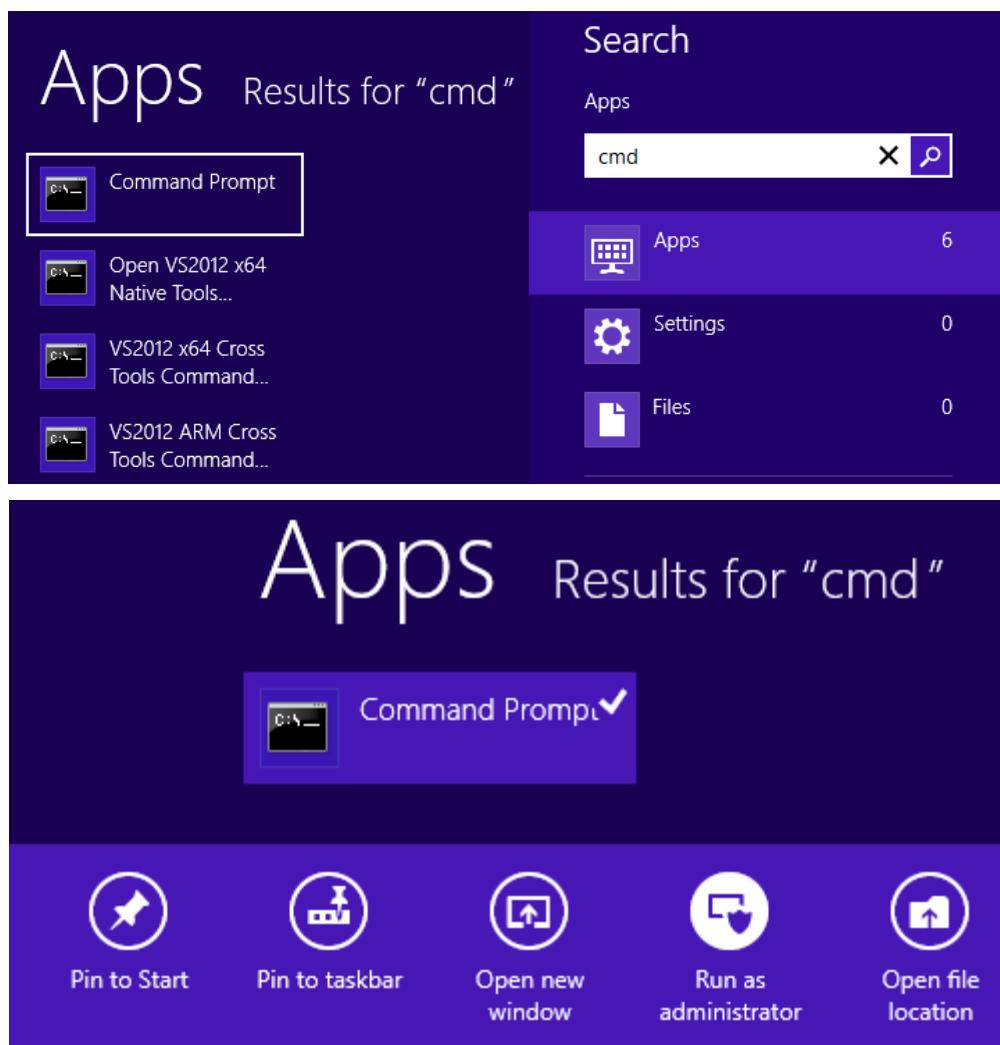
The time has come to **compile and execute** the simple example program written in C# we already discussed. To accomplish that, we need to do the following:

- Create a file named **HelloCSharp.cs**;
- Write the sample program in the file;
- Compile **HelloCSharp.cs** to an executable file **HelloCSharp.exe** using the console-based C# compiler (**csc.exe**);
- Execute the **HelloCSharp.exe** file.

Now, let's do it on the computer!

The instructions above vary depending on the **operating system**. Since programming on Linux is not the focus of this book, we will take a thorough look at what we need to write and execute the sample program on **Windows**. For those of you, who want to program in C# in a Linux environment, we already explained the Mono project, and you can download it and experiment.

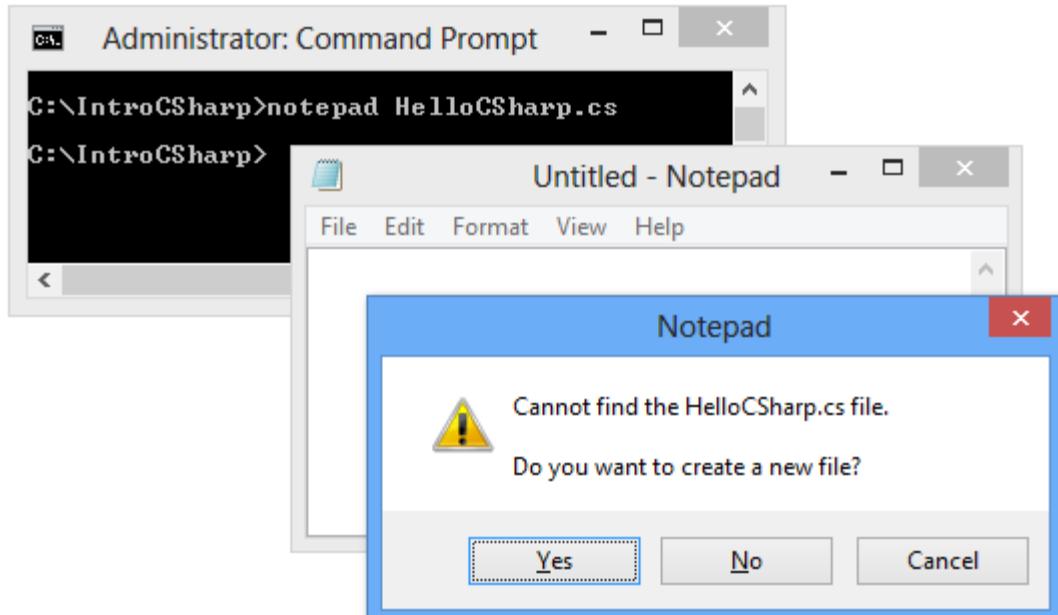
Here is the code of our **first C# program**:

HelloCSharp.cs

```
class HelloCSharp
{
   static void Main()
   {
      System.Console.WriteLine("Hello C#!");
   }
}
```

Creating C# Programs in the Windows Console

First we start the Windows command console, also known as **Command Prompt**. In **Windows 7** this is done from the Windows Explorer start menu: Start -> Programs -> Accessories -> Command Prompt.

It is advised that we **run the console as administrator** (right click on the **Command Prompt** icon and choose "**Run as administrator**"). Otherwise some operations we want to use may be restricted.

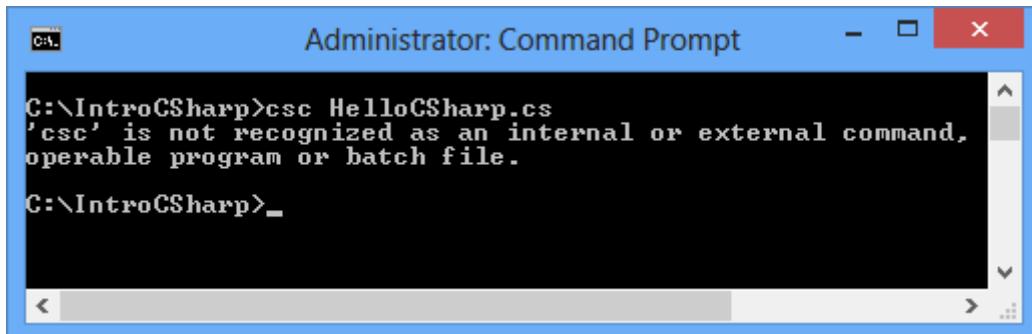
In **Windows 8** the "Run as administrator" command is directly available when you right click the command prompt icon from the Win8 Start Screen:

After opening the console, let's create a directory, in which we will experiment. We use the **md** command to create a directory and **cd** command to navigate to it (enter inside it):

The directory will be named **`IntroCSharp`** and will be located in **`C:\`**. We change the current directory to **`C:\IntroCSharp`** and create a new file **`HelloCSharp.cs`**, by using the built-in Windows text editor – Notepad.

To create the text file "**`HelloCSharp.cs`**", we execute the following command on the console:

```
notepad HelloCSharp.cs
```

This will start **Notepad** with the following dialog window, confirming the creation of a new file:

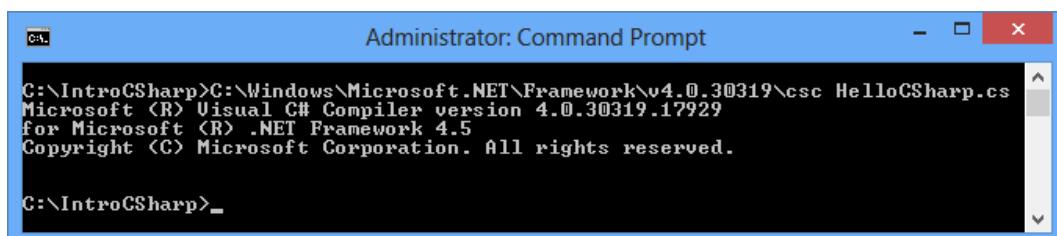
Notepad will warn us that **no such file exists** and will ask us if we want to create it. We click [Yes]. The next step is to rewrite or simply Copy / Paste the program's source code.

HelloCSharp.cs - Notepad

File Edit Format View Help

```
class HelloCSharp
{
	static void Main()
	{
		System.Console.WriteLine("Hello C#!");
	}
}
```

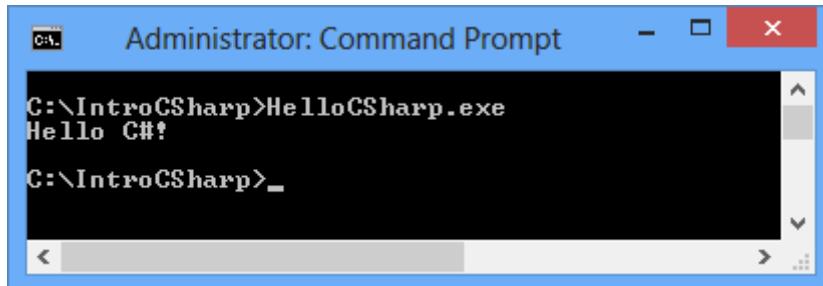
We save it by pressing [Ctrl+S] and close the Notepad editor with [Alt+F4]. Now we have the initial code of our sample C# program, written in the file **C:\IntroCSharp\HelloCSharp.cs**.

Compiling C# Programs in Windows

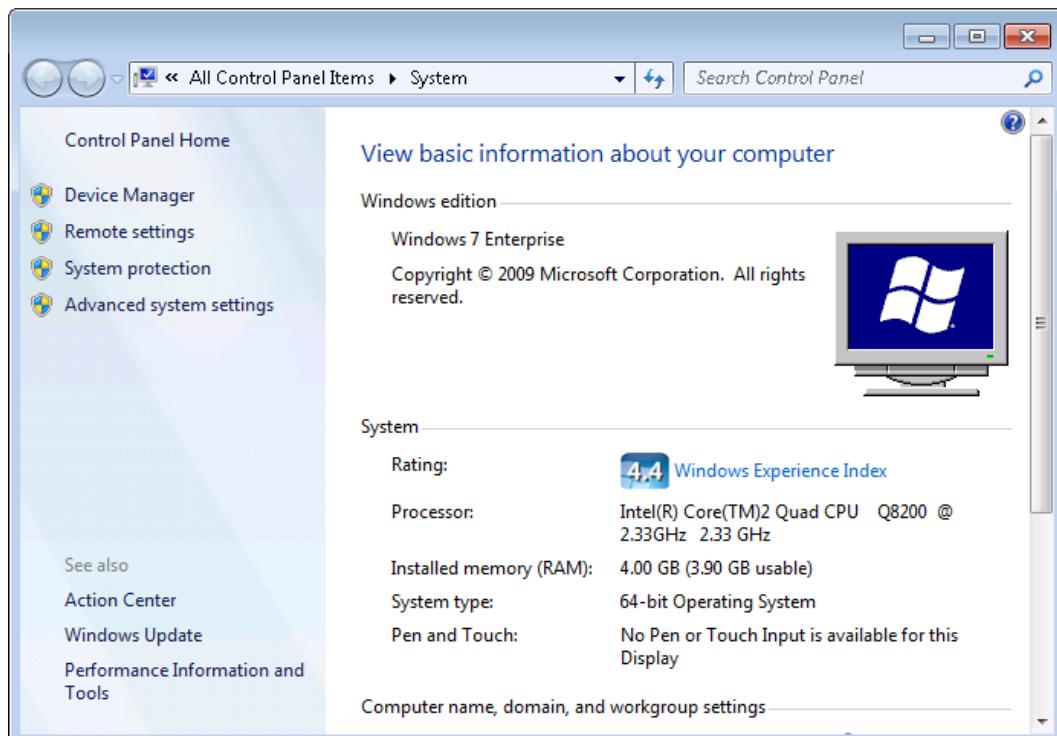
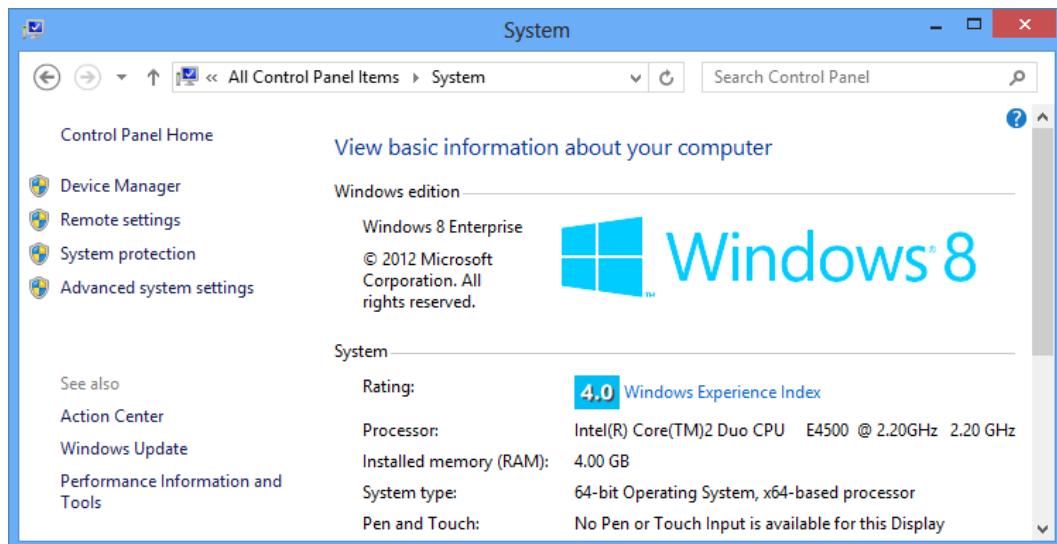
The only thing left to do is to compile and execute it. **Compiling** is done by the **csc.exe** compiler.

We got our first **error** – Windows cannot find an executable file or command with the name "**csc**". This is a very common problem and it is normal to appear if it is our first time using C#. Several reasons might have caused it:

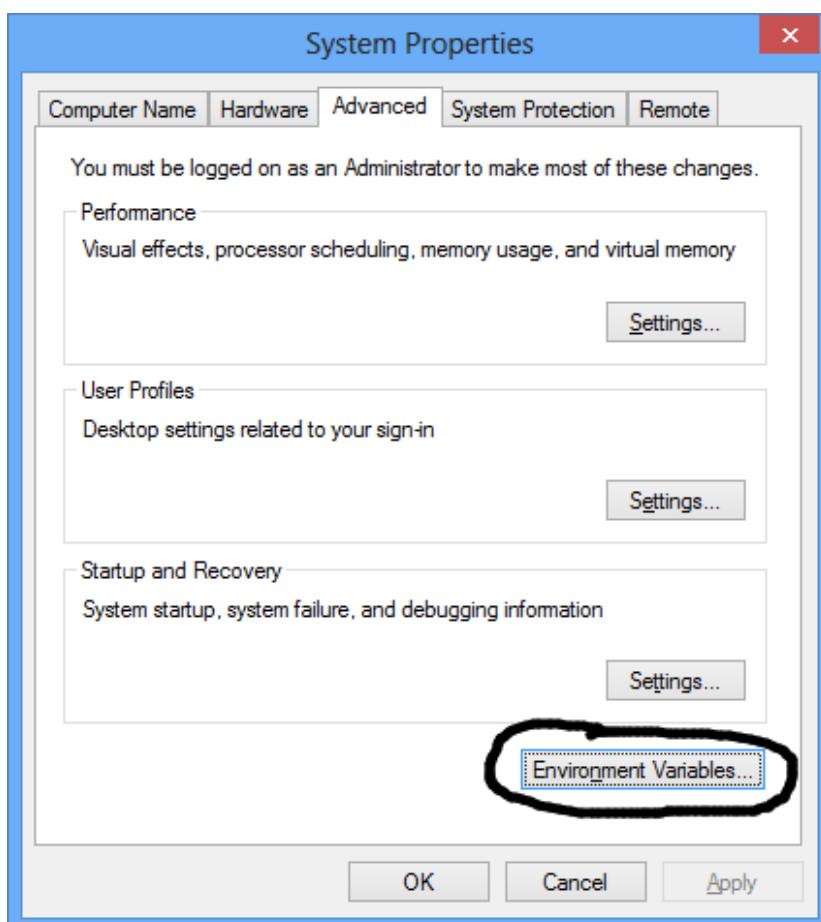
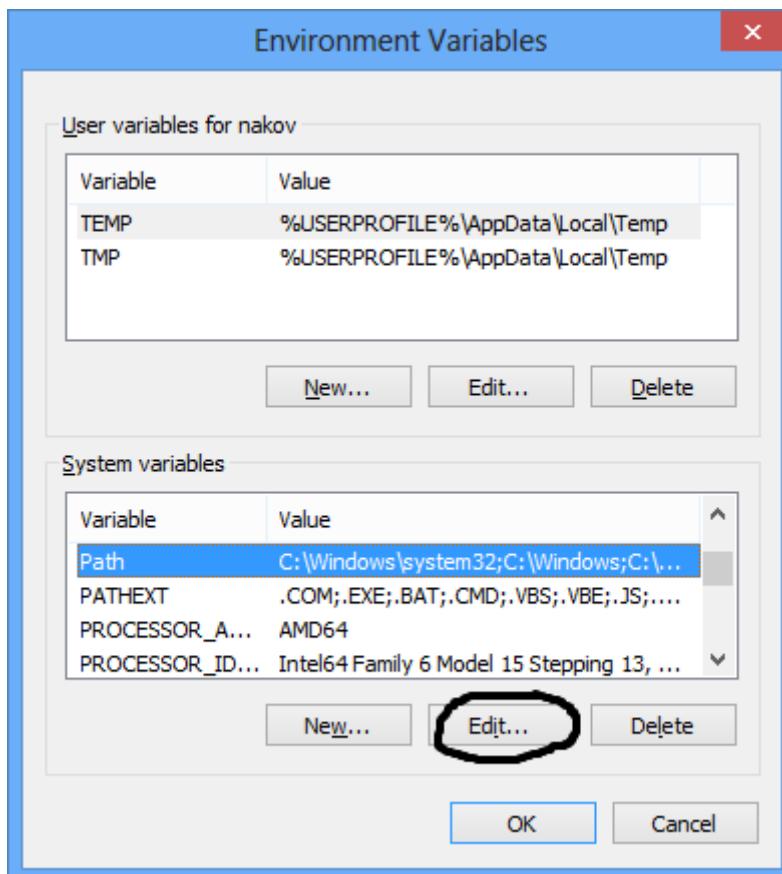
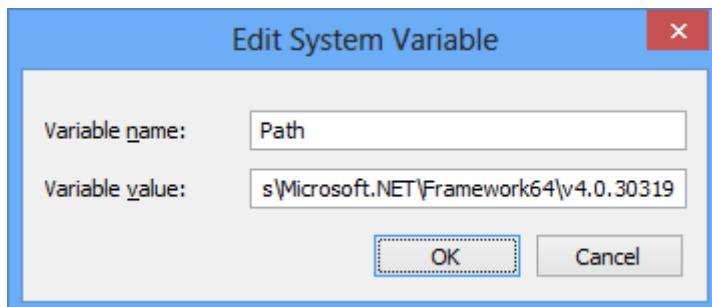
- The .NET Framework is not installed;
- The .NET Framework is installed correctly, but its directory **Microsoft.NET\Framework\v4.0.xxx** is not added to **the system path** for executable files and Windows cannot find **csc.exe**.

The first problem is easily solved by installing the .NET Framework (in our case – version 4.5). The other problem can be solved by changing the system path (we will do this later) or by using the full path to **csc.exe**, as it is shown on the figure below. In our case, the full file path to the C# compiler is **C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe** (note that this path could vary depending on the .NET framework version installed). Strange or not, **.NET 4.5** coming with Visual Studio 2012 and C# 5 installs in a directory named "**v4.0.30319**" – this is not a mistake.

Compiling and Running C# Programs in Windows

Now let's invoke the **csc** compiler through its full path and pass to it the file we want to compile as a parameter (**HelloCSharp.exe**):

```
Administrator: Command Prompt
C:\IntroCSharp>C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc HelloCSharp.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\IntroCSharp>_
```

After the execution **csc** is completed without any errors, and we get the following file as a result: **C:\IntroCSharp\HelloCSharp.exe**. To run it, we simply need to write its name. The result of the execution of our program is the message "Hello, C#!" printed on the console. It is not great but it is a good start:

Changing the System Paths in Windows

If we know to use the command line C# compiler (**csc.exe**) without entering the full path to it, we could add its folder to the **Windows system path**.

1. We open **Control Panel** and select "**System**". As a result this well-known window appears (the screenshot is taken from **Windows 7**):

In **Windows 8** it might look a bit different, but is almost the same:

2. We select "**Advanced system settings**". The dialog window "**System Properties**" appears:

System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

System startup, system failure, and debugging information

Settings...

Environment Variables...

OK Cancel Apply

3. We click the button "**Environment Variables**" and a window with all the environment variables shows up:

4. We choose "**Path**" from the list of **System variables**, as shown on the figure, and press the "Edit" button. A small window appears, in which we enter the path to the directory where the .NET Framework is installed:

Of course, first we need to find where our .NET Framework is installed. By default it is located somewhere inside the Windows system directory **`C:\Windows\Microsoft.NET`**, for example:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319
```

Adding the additional path to the already existing ones in the **Path** variable of the environment is done by adjoining the path name to the others and using a semicolon (**;**) as a spacer.

We must be careful because if we delete any of the existing system paths, some of Windows' functions or part of the installed software might fail to operate properly!

5. When we are done with **setting the path,** we can try running **csc.exe**, without entering its full path. To do so, we open a new **cmd.exe** (Command Prompt) window (it is important to **restart the Command Prompt**) and type in the "**csc**" command. We should see the C# compiler version and a message that no input file has been specified:

Administrator: Command Prompt

```
C:\IntroCSharp>csc
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

warning CS2008: No source files specified
error CS1562: Outputs without source must have the /out optio

C:\IntroCSharp>_
```

Visual Studio IDE

So far we have examined how to compile and run C# programs using the **Windows console** (Command Prompt). Of course, there is an easier way to do it – by using an integrated development environment, which will execute all the commands we have used so far. Let's take a look at how to work with **development environments (IDE)** and how they will make our job easier.

Integrated Development Environments

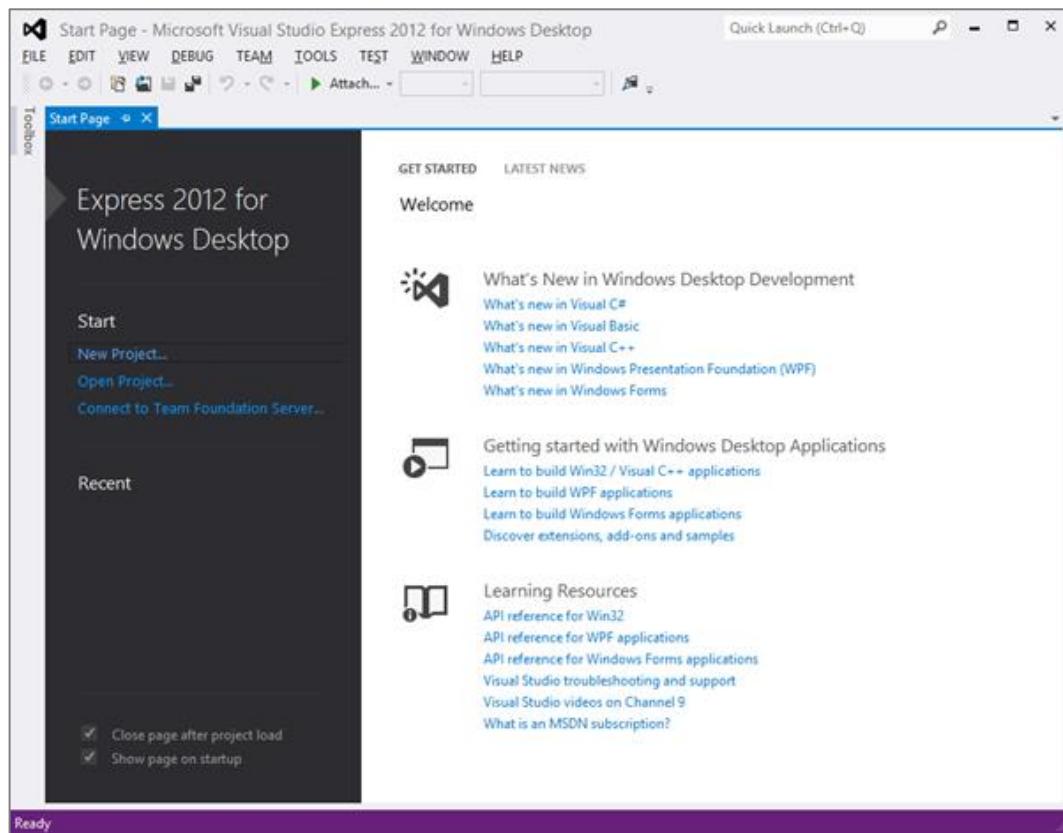
In the previous examples, we examined how to compile and run a program consisting of a single file. Usually programs are made of many files, sometimes even tens of thousands. Writing in a text editor, compiling and executing a single file program from the command prompt are simple, but to do all this for a big project can prove to be a very complex and time-consuming endeavor. There is a **single tool** that reduces the complexity, makes writing, compiling and executing software applications easier – the so called **Integrated Development Environment** (**IDE**). Development environments usually offer many additions to the main development functions

such as debugging, unit testing, checking for common errors, access to a repository and others.

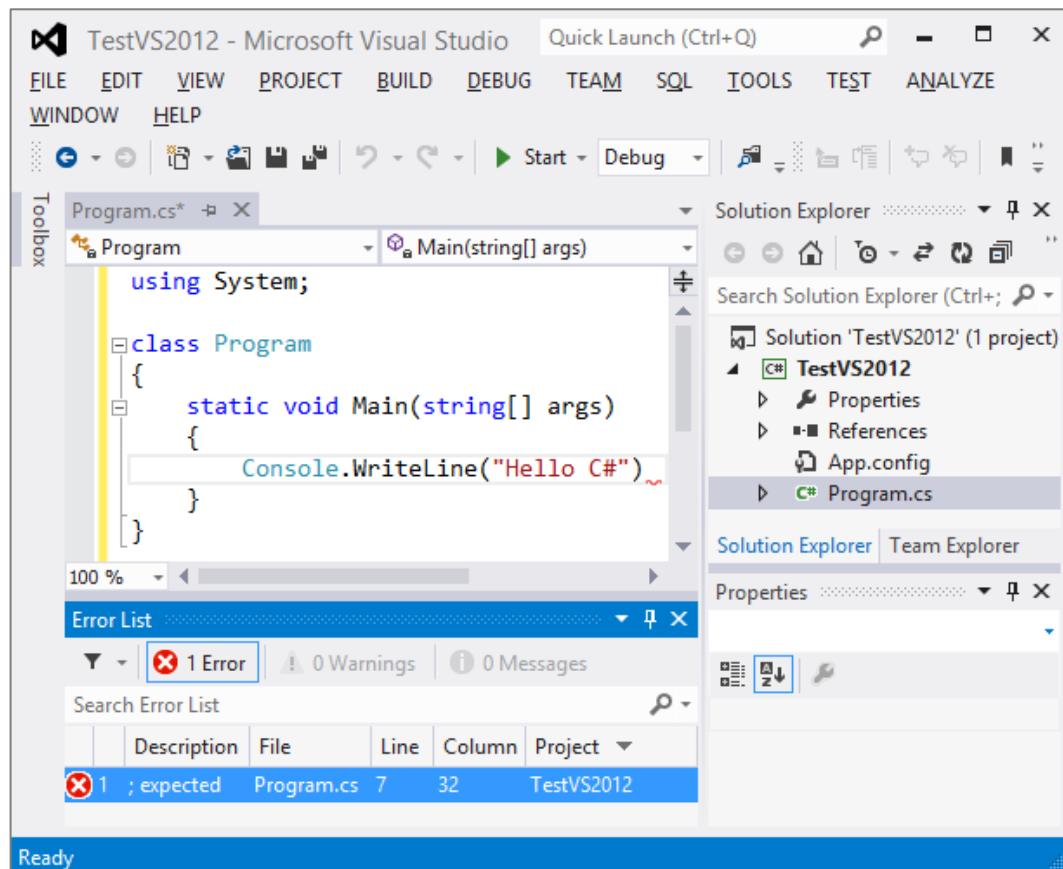
What Is Visual Studio?

Visual Studio is a powerful integrated environment (**IDE**) for developing software applications for Windows and the .NET Framework platform. Visual Studio (VS) supports **different programming languages** (for example C#, VB.NET and C++) and **different software development technologies** (Win32, COM, ASP.NET, ADO.NET Entity Framework, Windows Forms, WPF, Silverlight, Windows Store apps and many more Windows and .NET technologies). It offers a powerful integrated environment for **writing code**, **compiling**, **executing**, **debugging** and testing applications, designing user interface (forms, dialogs, web pages, visual controls and others), data and class modeling, running tests and hundreds of other functions.

IDE means "integrated development environment" – a tool where you write code, compile it, run it, test it, debug it, etc. and **everything is integrated** into a single place. Visual Studio is typical example of development IDE.

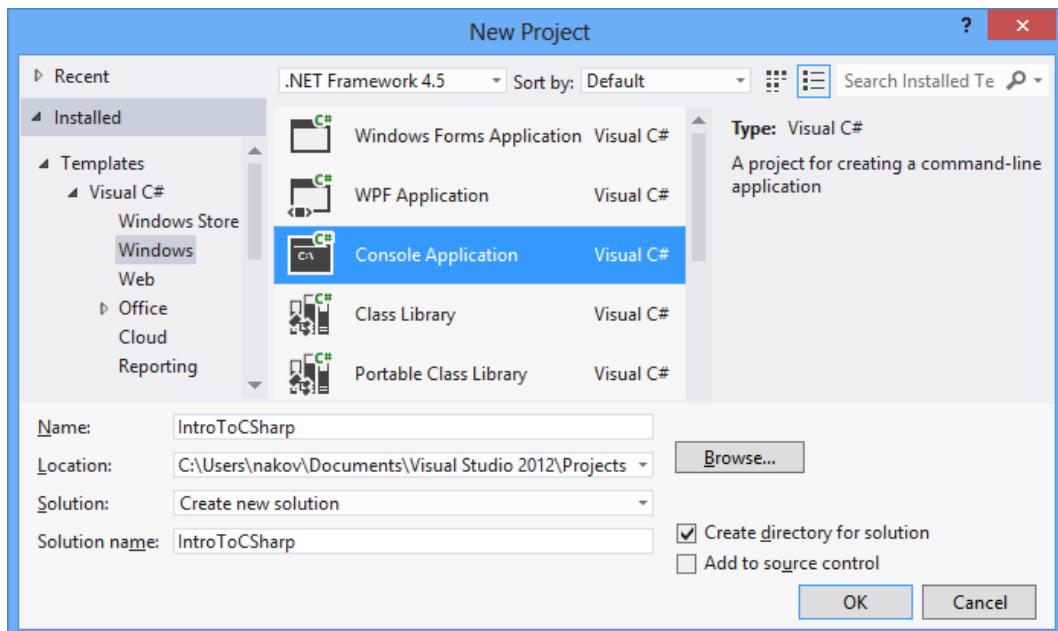
.NET Framework 4.5 comes with **Visual Studio 2012** (VS 2012). This is the latest version of Visual Studio as of March 2013. It is designed for **C# 5**, **.NET 4.5** and **Windows 8** development.

VS 2012 is a commercial product but has a free version called **Visual Studio Express 2012**, which can be downloaded for free from the Microsoft website at http://microsoft.com/visualstudio/downloads.

Visual Studio 2012 Express has several editions (for Desktop, for Web, for Windows 8 and others). If you want to write C# code following the content of this book, you may use **Visual Studio 2012 Express for Desktop** or check whether you have a free license of the full Visual Studio from your University or organization. Many academic institutions (like Sofia University and Telerik Software Academy) provide free Microsoft **DreamSpark accounts** to their students to get licensed Windows, Visual Studio, SQL Server and other development tools. If you are student, ask your university administration about the DreamSpark program. Most universities worldwide are members of this program.

In this book we will take a look at only the most important functions of **VS Express 2012** – the ones related to coding. These are the functions for creating, editing, compiling, executing and debugging programs.

Note that older Visual Studio versions such as **VS 2010** and **VS 2008** can also be used for the examples in this book but their user interface might look slightly different. Our examples are based on **VS 2012 on Windows 8**.

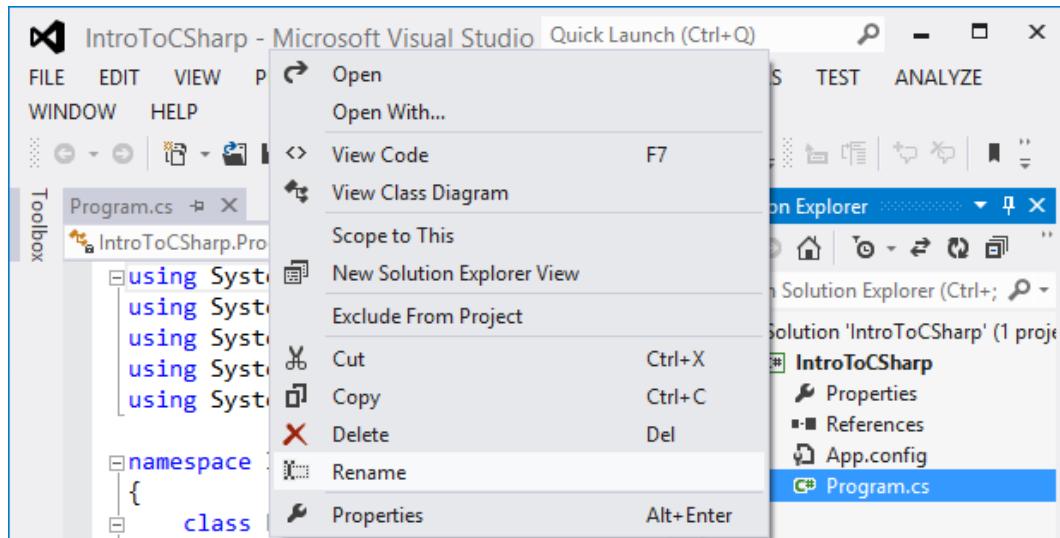
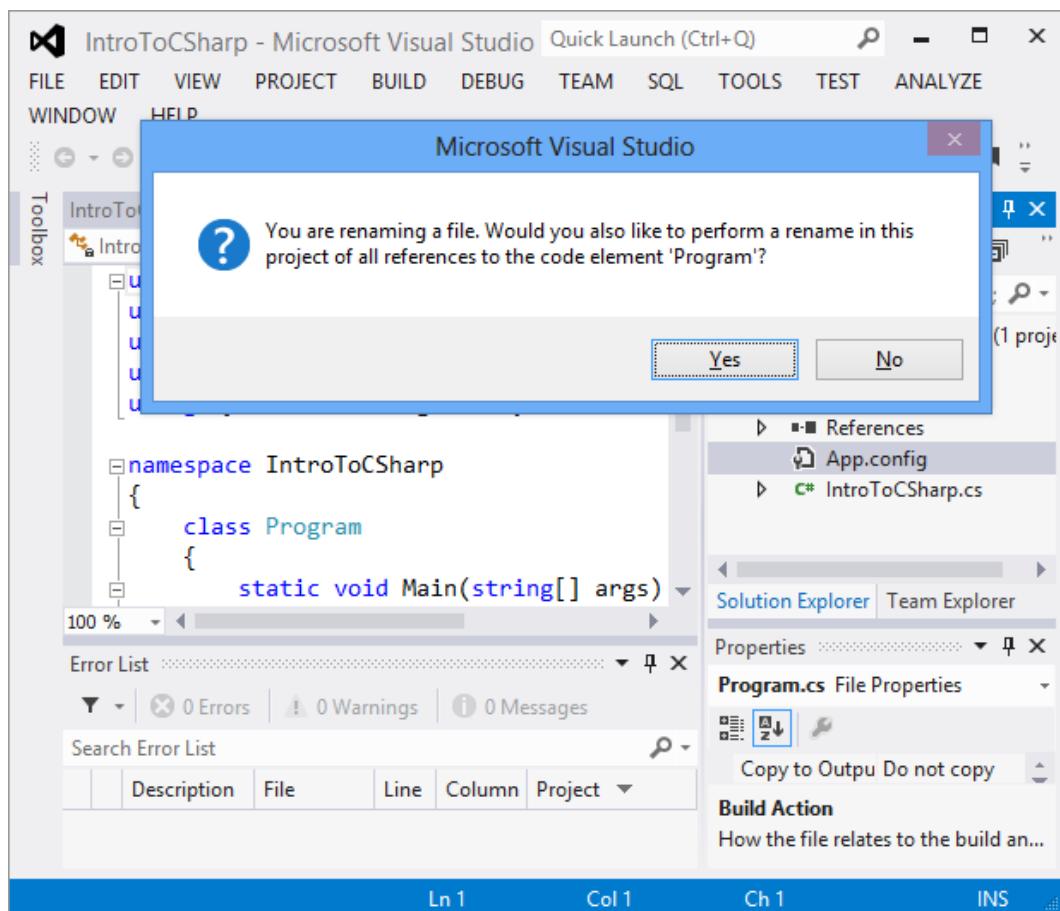
Before we continue with an example, let's take a more detailed look of the structure of **Visual Studio 2012's** visual interface. Windows are the main part of it. Each of them has a different function tied to the development of applications. Let's see how **Visual Studio 2012** looks after the default installation and configuration:

Visual Studio has several windows that we will explore (see the figures above and below):

- **Start Page** – from the start page we can easily open any of our latest projects or start a new one, to create our first C# program or to get help how to use C#.
- **Code Editor** – keeps the program's source code and allows opening and editing multiple files.
- **Error List** – it shows the errors in the program we develop (if any). We learn how to use this window later when we compile C# programs in Visual Studio.
- **Solution Explorer** – when no project is loaded, this window is empty, but it will become a part of our lives as C# programmers. It will show the structure of our project – all the files it contains, regardless if they are C# code, images or some other type of code or resources.
- **Properties** – holds a list of the current object's properties. Properties are used mainly in the component-based programming, e.g. when we develop WPF, Windows Store or ASP.NET Web Forms application.

There are many other windows with auxiliary functionality in Visual Studio but we will not review them at this time.

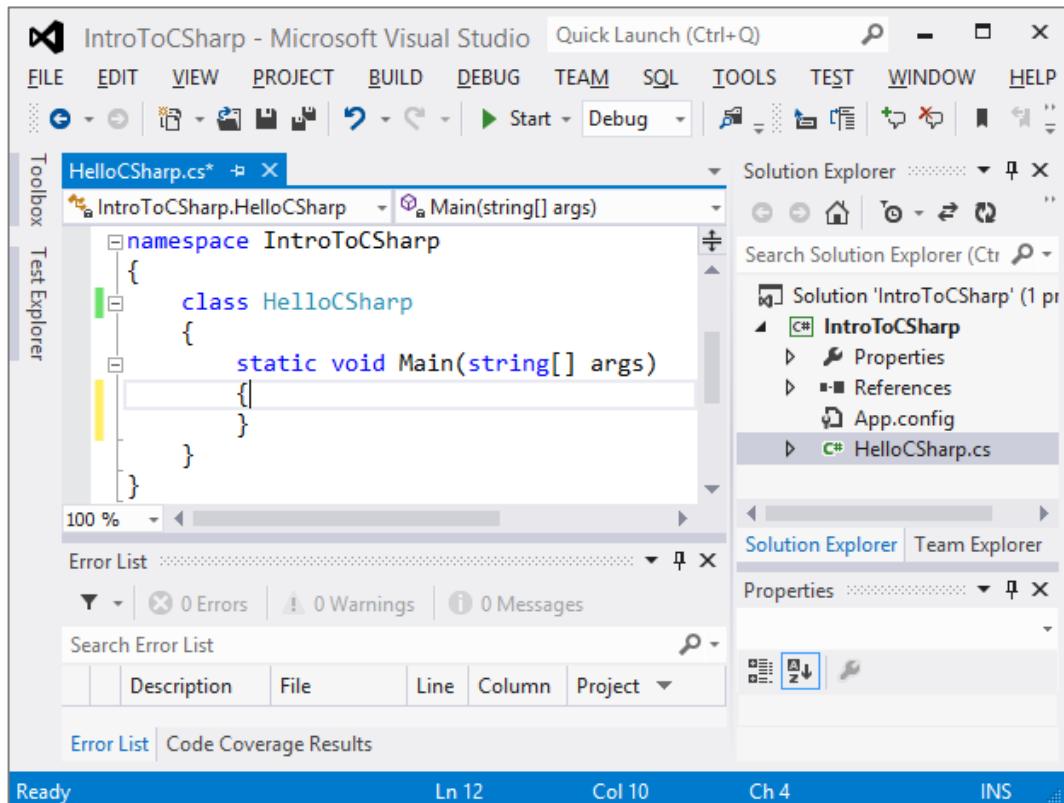
Creating a New C# Project

Before doing anything else in Visual Studio, we must **create a new project** or load an existing one. The project groups many files, designed to implement a software application or system, in a logical manner. It is recommended that we create a separate project for each new program.

We can **create a project in Visual Studio** by following these steps:

- **File** -> **New Project** …
- The "New Project" dialog appears and lists all the different types of projects we can create. We can choose a **project type** (e.g. Console Application or WPF Application), **programming language** (e.g. C# or VB.NET) and **.NET Framework version** (e.g. .NET Framework 4.5) and give a name to our project (in our case "**IntroToCSharp**"):

- We choose **Console Application**. Console applications are programs, which use the console as a default input and output. Data is entered with the keyboard and when a result needs to be printed it appears on the console (as text on the screen in the program window). Aside from console applications, we can create applications with a graphical user interface (e.g. Windows Forms or WPF), Web applications, web services, mobile applications, Windows Store apps, database projects and others.

- In the field "Name" we enter the name of the project. In our case we choose the name **`IntroToCSharp`**.

- We press the **[OK]** button.

The newly created project is now shown in the **Solution Explorer**. Also, our first file, containing the program code, is automatically added. It is named **`Program.cs`**. It is very important to **give meaningful names** to our files, classes, methods and other elements of the program, so that we can easily find them and navigate the code. A **meaningful name** means a name that answers the question "what is the intent of this file / class / method / variable?" and helps developers to understand how the code works. Don't use **`Problem3`** for a name, even if you are solving the problem 3 from the exercises. Name your project / class by its **purpose**. If your project is well named, after few months or a year you will be able to explain what it is intended to do without opening it and looking inside. **`Problem3`** says nothing about what this project actually does.

In order to rename the **`Program.cs`** file, we right click on it in the Solution Explorer and select "Rename". We can name the main file of our C# program **`HelloCSharp.cs`**. Renaming a file can also be done with the [F2] key when the file is selected in the Solution Explorer:

A dialog window appears asking us if we want to rename class name as well as the file name. We select "**Yes**".

After we complete all these steps we have our first console application named **IntroToCSharp** and containing a single class **HelloCSharp** (stored in the file **HelloCSharp.cs**):

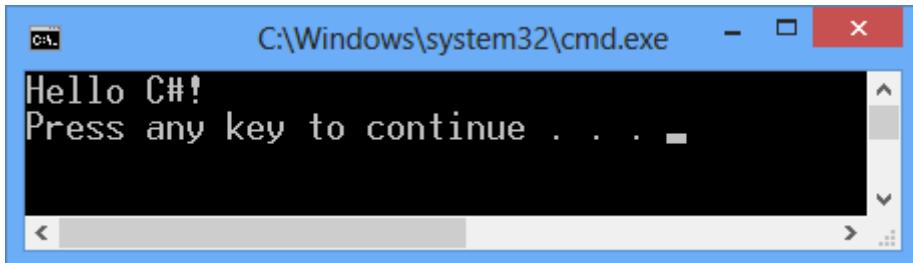
All we have to do is **add code to the Main() method**. By default, the **HelloCSharp.cs** code should be loaded and ready for editing. If it is not, we double click on the **HelloCSharp.cs** file in the Solution Explorer to load it. We enter the following source code:

```
namespace IntroToCSharp
{
    class HelloCSharp
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello C!");
        }
    }
```

Compiling the Source Code

The compiling process in Visual Studio includes several steps:

- Syntax error check;

- A check for other errors, like missing libraries;
- Converting the C# code into an executable file (a .NET assembly). For console applications it is an **.exe** file.

To **compile** a file in Visual Studio, we press the **[F6]** key or **[Shift+Ctrl+B]**. Usually, errors are underlined in red, to attract the programmer's attention, while we are still writing or when compiling, at the latest. They are listed in the "Error List" window if it is visible (if it is not, we can show it from the "View" menu of Visual Studio).

If our project has at least one error, it will be marked with a small red "**x**" in the "**Error List**" window. Short info about the problem is displayed for each error – filename, line number and project name. If we double click any of the errors in the "Error List", Visual Studio will automatically take us to the file and line of code where the error has occurred. In the screenshot above the problem is that we have "**using Systema**;" instead of "**using System**".

Starting the Project

To start the project, we press **[Ctrl+F5]** (holding the [Ctrl] key pressed and at the same time pressing the [F5] key).

The program will start and the result will be displayed on the console, followed by the "**Press any key to continue . . .**" message:

The last message is not part of the result produced by the program. It is a reminder by Visual Studio that **our program has finished its execution** and it gives us time to see the result. If we run the program by only pressing **[F5]**, that message will not appear and the result will vanish instantly after appearing because the program will have finished its execution, and the window will be closed. That is why we should **always start our console applications by pressing [Ctrl+F5]**.

Not all project types can be executed. In order to execute a C# project, it needs to have one class with a **`Main()`** method declared in the way described earlier in this chapter.

Debugging the Program

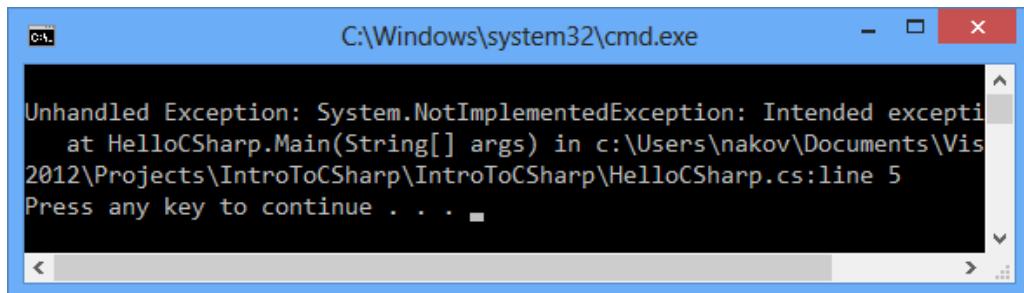
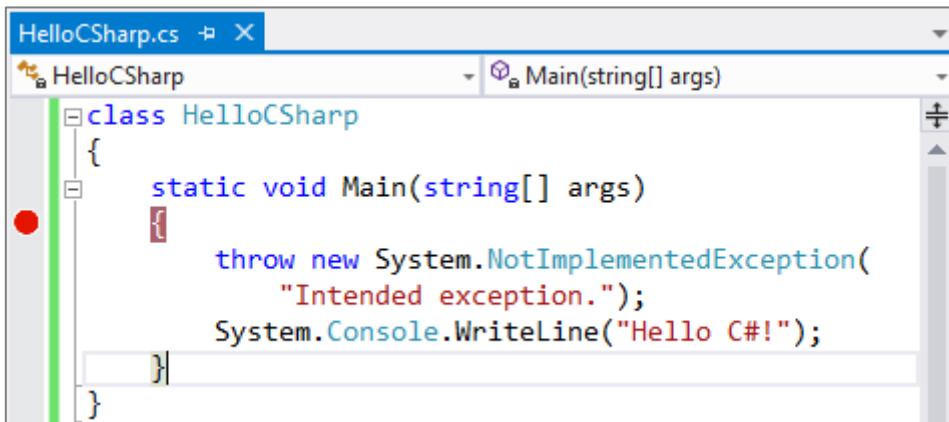
When our program contains errors, also known as **bugs**, we must find and remove them, i.e. we need to **debug** the program. The debugging process includes:

- Noticing the **problems** (bugs);
- Finding the code **causing** the problems;
- **Fixing** the code so that the program works correctly;
- **Testing** to make sure the program works as expected after the changes are made.

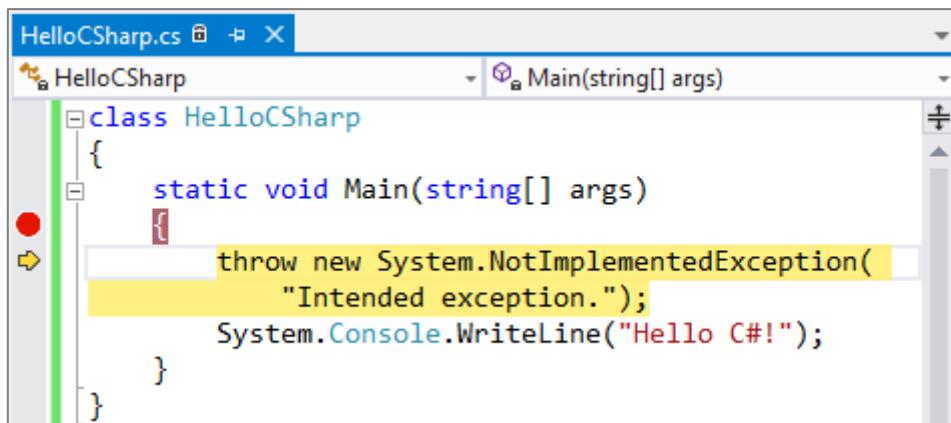
The process can be repeated several times until the program starts working correctly. After we have noticed the problem, we need to find the code causing it. Visual Studio can help by allowing us to check **step by step** whether everything is working as planned.

To stop the execution of the program at designated positions we can place **breakpoints**. The breakpoint is associated with a line of the program. The program **stops its execution** on the lines with breakpoints, allowing for the rest of the code to be executed step by step. On each step we can check and even change the values of the current variables.

Debugging is a sort of **step by step** slow motion execution of the program. It gives us the opportunity to easily understand the details of the code and see where exactly and why the errors have occurred.

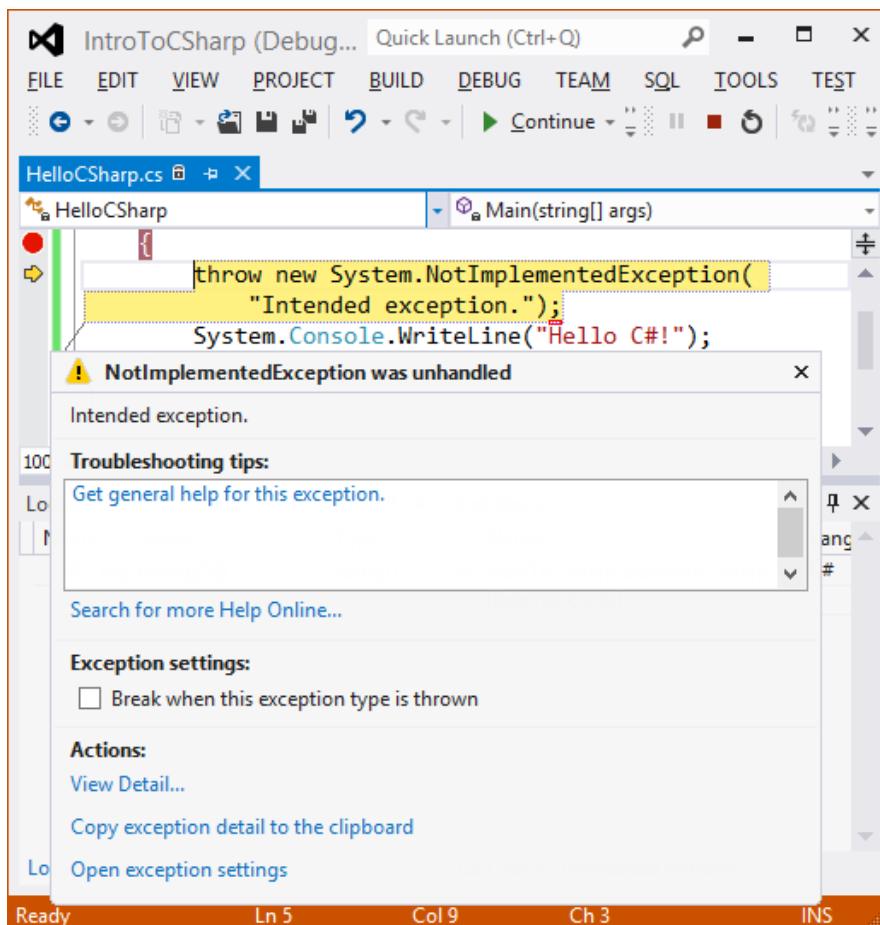
Let's create an **intentional error in our program**, to illustrate how to use breakpoints. We will add a line to the program, which will create an exception during the execution (we will take a detailed look at exceptions in the "Exception Handling" chapter).

For now let's edit our program in the following way:

HelloCSharp.cs
`class HelloCSharp`

```
{
   static void Main()
   {
      throw new System.NotImplementedException(
         "Intended exception.");
      System.Console.WriteLine("Hello C#!");
   }
}
```

When we start the program again with **[Ctrl+F5]** we will get an error and it will be printed on the console:

Let's see how **breakpoints will help us** find the problem. We move the cursor to the line with the opening bracket of the **Main()** method and press [F9] (by doing so we place a breakpoint on that line). A red dot appears, indicating that the program will stop there if it is executed in debug mode:

```
HelloCSharp.cs
HelloCSharp                Main(string[] args)
class HelloCSharp
{
    static void Main(string[] args)
    {
        throw new System.NotImplementedException(
            "Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}
```

Now we must start the program in debug mode. We select **Debug** -> **Start Debugging** or press **[F5]**. The program will start and immediately stop at the first breakpoint it encounters. The line will be colored in yellow and we can execute the program step by step. With the [F10] key we move to the next line.

When we are on a given line and it is colored in **yellow**, the code on that line is **not executed yet**. It executes once we have passed that line. In this case

we have not received the error yet despite the fact that we are on the line we added and should cause it:

HelloCSharp.cs — HelloCSharp — Main(string[] args)

```
class HelloCSharp
{
    static void Main(string[] args)
    {
        throw new System.NotImplementedException(
            "Intended exception.");
        System.Console.WriteLine("Hello C#!");
    }
}
```

We press **[F10]** one more time to execute the current line. This time Visual Studio displays a window specifying the line, where the error occurred as well as some additional details about it:

Once we know where exactly the problem in the program is, we can easily correct it. To do so, first, we need to stop the execution of the program before it is finished. We select **Debug** -> **Stop Debugging** or press [Shift+F5]. After that we delete the problem line and start the program in normal mode (without debugging) by pressing) [Ctrl+F5].

Alternatives to Visual Studio

As we have seen, in theory, we can do without Visual Studio, but in practice that is not a good idea. The work required compiling a big project, finding all the errors in the code and performing numerous other actions would simply take too much time without Visual Studio.

On the other hand, **Visual Studio is not a free** software developing environment (the full version). Many people cannot afford to buy the professional version (this is also true for small companies and some people engaged in programming).

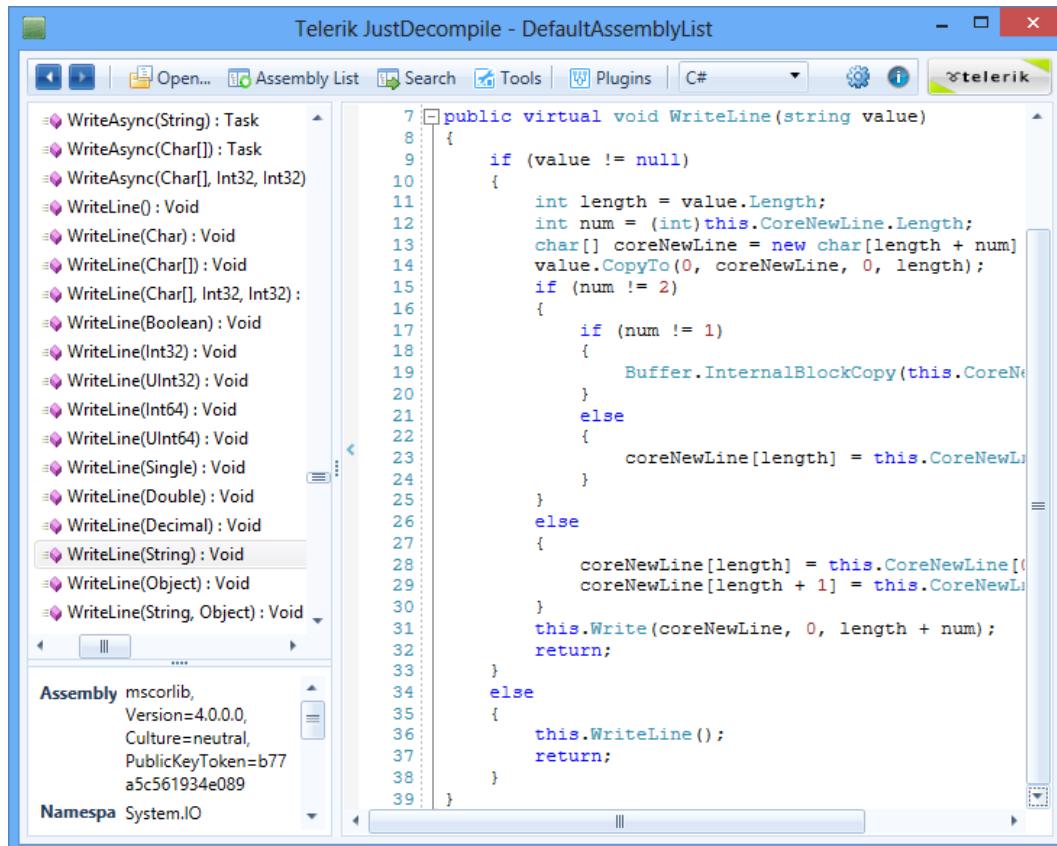
This is why there are some alternatives to Visual Studio (except VS Express Edition), which are free and can handle the same tasks relatively well.

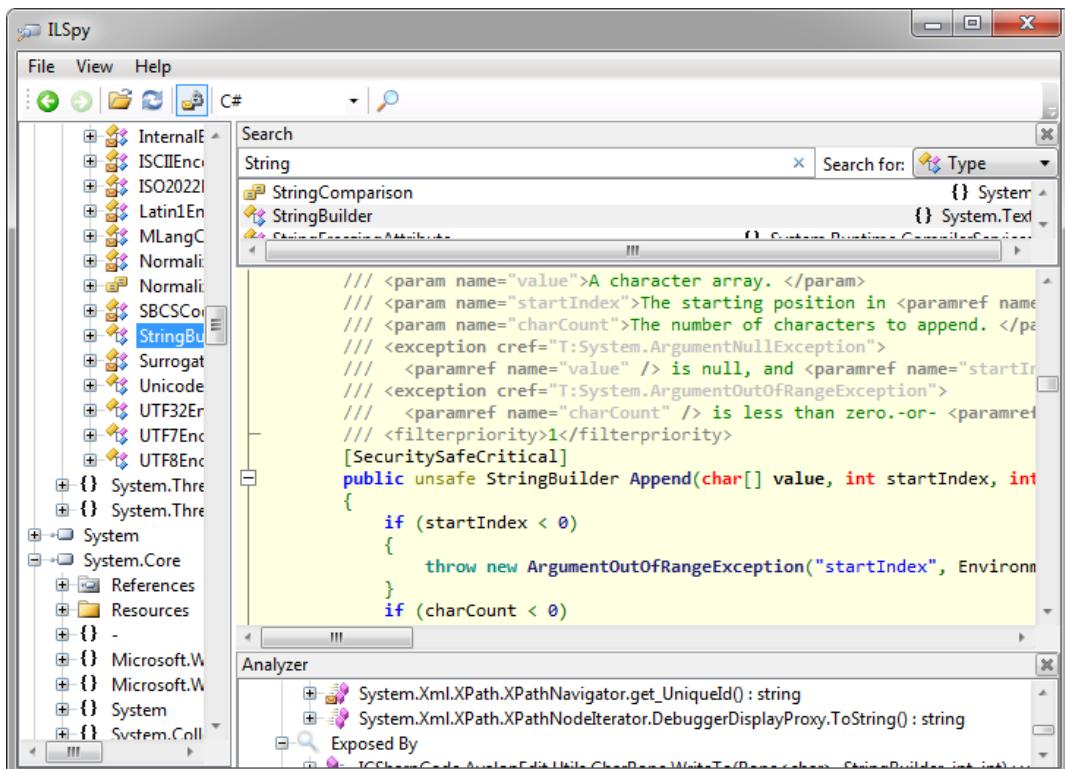
SharpDevelop

One alternative is **SharpDevelop** (**#Develop**). We can find it at the following Internet address: http://www.icsharpcode.NET/OpenSource/SD/. #Develop is an IDE for C# and is developed as an open-source project. It supports the majority of the functionalities offered in Visual Studio 2012 but also works in Linux and other operating systems. We will not review it in details but you should keep it in mind, in case you need a C# development environment and Visual Studio is not available.

MonoDevelop

MonoDevelop is an integrated software development environment for the .NET platform. It is completely free (open source) and can be downloaded at: http://monodevelop.com. With MonoDevelop, we can quickly and easily write fully functional desktop and ASP.NET applications for Linux, Mac OS X and Windows. It also enables programmers to easily transfer projects created in Visual Studio to the Mono platform and make them functional in other platforms.

Decompiling Code

Sometimes programmers need to see the code of a given module or program, not written by them and with no source code available. The process, which **generates source code from an existing executable binary fi**le (.NET assembly – **.exe** or **.dll**) is called **decompiling**.

We might need to decompile code in the following cases:

- We want to check how a given **algorithm** is implemented but we do not have the source code, e.g. to check how **`Array.Sort()`** internally works.

- There are several options when using some .NET library, and we want to find the optimal choice. We want to see **how to use certain API** digging into some compiled code that uses it.

- We have no information **how a given library works**, but we have the compiled code (.NET assembly), which uses it, and we want to find out how exactly the library works.

- We have lost our source code and we want to recover it. **Code recovery** through decompilation will result in lost variable names, comments, formatting, and others, but is better than nothing.

Decompiling is done with the help of tools, which are not standard part of Visual Studio. The first popular **.NET decompiler** was Red Gate's **Reflector** (before it became commercial in early 2011).

Telerik is offering a good and completely free .NET decompiler called **JustDecompile**. It can be downloaded from the company's website: http://www.telerik.com/products/decompiler.aspx. JustDecompile allows code decompilation directly in Visual Studio and also has an external stand-alone GUI application for browsing assemblies and decompile their code:

Another good decompilation tool for .NET is the **ILSpy**, which is developed around the SharpDevelop project. ILSpy can be downloaded at: http://ilspy.net. The program does not require installation. After we start it, ILSpy loads some of the standard .NET Framework libraries. Via the menu File -> Open, we can open a certain .NET assembly. We can also load an assembly from the GAC (Global Assembly Cache). This is how ILSpy looks like:

In ILSpy there are two ways to find out how a given method is implemented. For example, if we want to see how the static method **`System.Currency.ToDecimal`** works, first we can use the tree on the left to find the **`Currency`** class in the **`System`** namespace and finally select the **`ToDecimal`** method. If we click on any method, we will be able to see its source code in C#. Another way to find a given class is using the search engine in ILSpy. It searches through the names of all classes, interfaces, methods, properties etc. from the loaded assemblies. Unfortunately, the version at the time of writing of this book (ILSpy 2.1) can decompile only the languages C#, VB.NET and IL.

JustDecompile and ILSpy are **extremely useful tools**, which can help almost every day when developing .NET software and we should definitely download at least one and play with it. When we are wondering how a certain method works or how something is implemented in a given assembly, we can always rely on the decompiler to find out.

C# in Linux, iOS and Android

C# programming in Linux is not very developed compared to that in Windows. We do not want to completely skip it, so we will give some guidelines on how to start **programming in C# in Linux, iOS and Android**.

The most important thing that we need in order to write C# code in Linux is a .NET Framework implementation. Microsoft .NET Framework is not available for Linux but there is an **open-source .NET implementation called "Mono"**. We can download Mono at its official website: http://www.mono-project.com. Mono allows us to compile and execute C# programs in a Linux environment and on other operating systems. It contains a C# compiler, a CLR, a garbage collector, the standard .NET libraries and many of the libraries available for .NET Framework in Windows like Windows Forms and ASP.NET.

Mono supports compiling and running C# code not only in **Linux** but also in **Solaris, Mac OS X, iOS** (iPhone / iPad) and **Android**. The iOS version (MonoTouch) and the Android version of Mono (Mono for Android) are commercial projects, while Mono for Linux is open-source free software.

Of course, Visual Studio does not work in Linux environment but we can use the #Develop or MonoDevelop as C# IDE in Linux.

Other .NET Languages

C# is the most popular .NET language but there are few other languages that may be used to write .NET programs:

- **VB.NET** – Visual Basic .NET (VB) is Basic language adapted to run in .NET Framework. It is considered a successor of Microsoft Visual Basic 6 (legacy development environment for Windows 3.1 and Windows 95). It has strange syntax (for C# developers) but generally does the same as C#, just in different syntax. The only reason VB.NET exists is historical: it is successor of VB6 and keeps most of its syntax. **Not recommended** unless you are VB6 programmer.

- **Managed C++** – adaptation of the C++ programming language to .NET Framework. It can be useful if you need to quickly convert existing C++ code to be used from .NET. Not recommended for new projects. **Not recommended** for the readers of this book, even if someone has some C++ experience, because it makes .NET programming unnecessary complicated.

- **F#** – an experiment to put purely functional programming paradigm in .NET Framework. **Not recommended** at all (unless you are functional programming guru).

- **JavaScript** – it may be used to develop Windows 8 (Windows Store) applications through the **WinJS** technology. It might be a good choice for skillful HTML5 developers who have good JavaScript skills. **Not recommended** for the readers of this book because it does not support Console applications.

Exercises

1. Install and make yourself familiar with **Microsoft Visual Studio** and Microsoft Developer Network (**MSDN**) Library Documentation.
2. Find the description of the **System.Console** class in the standard .NET API documentation (MSDN Library).
3. Find the description of the **System.Console.WriteLine()** method and its different possible parameters in the MSDN Library.
4. **Compile and execute** the sample program from this chapter using the command prompt (the console) and Visual Studio.
5. **Modify** the sample program to print a different greeting, for example "Good Day!".
6. Write a console application that **prints your first and last name** on the console.
7. Write a program that **prints the following numbers** on the console 1, 101, 1001, each on a new line.
8. Write a program that prints on the console the **current date and time**.
9. Write a program that prints the **square root of 12345**.
10. Write a program that prints the first 100 members of the **sequence** 2, -3, 4, -5, 6, -7, 8.
11. Write a program that reads your age from the console and prints your **age after 10 years**.
12. Describe the difference between **C#** and the **.NET Framework**.
13. Make a list of the **most popular programming** languages. How are they different from C#?
14. **Decompile** the example program from exercise 5.

Solutions and Guidelines

1. If you have a **DreamSpark account** (www.dreamspark.com), or your school or university offers free access to Microsoft products, install the full version of **Microsoft Visual Studio**. If you do not have the opportunity to work with the full version of Microsoft Visual Studio, you can download **Visual Studio Express** for free from the Microsoft web site; it is completely free and works well for educational purposes.
2. Use the address given in the "**.NET Documentation**" section of this chapter. Open it and search in the tree on the left side. A **Google search** will work just as well and is often the fastest way to find documentation for a given .NET class.
3. Use **the same approach** as in the previous exercise.

4. Follow the instruction from the Compiling and Executing C# Programs section.

5. Use the code from the **sample C# program** from this chapter and change the printed message.

6. Find out how to use the **System.Console.Write()** method.

7. Use the **System.Console.WriteLine()** method.

8. Find out what features are offered by the **System.DateTime** class.

9. Find out what features are offered by the **System.Math** class.

10. Try to learn on your own how to use **loops** in C#. You may read about **for**-loops in the chapter "Loops".

11. Use the methods **System.Console.ReadLine()**, **int.Parse()** and **System.DateTime.AddYears()**.

12. **Research them** on the Internet (e.g. in **Wikipedia**) and take a closer look at the differences between them. You will find that **C#** is a programming language while **.NET Framework** is development platform and runtime for running .NET code. Be sure to read the section "**The C# Language and the .NET Platform**" form this chapter.

13. Find out which are the most popular languages and examine some sample programs written in them. Compare them to C#. You might take a look at **C**, **C++**, **Java**, **C#**, **VB.NET**, **PHP**, **JavaScript**, **Perl**, **Python** and **Ruby**.

14. First download and **install** **JustDecompile** or ILSpy (more information about them can be found in the "Code Decompilation" section). After you run one of them, open your program's compiled file. It can be found in the **bin\Debug** subdirectory of your C# project. For example, if your project is named **TestCSharp** and is located in **C:\Projects**, then the compiled assembly (executable file) of your program will be the following file **C:\Projects\TestCSharp\bin\Debug\TestCSharp.exe**.

Chapter 2. Primitive Types and Variables

In This Chapter

In this chapter we will get familiar with **primitive types and variables in C#** – what they are and how to work with them. First we will consider the **data types** – integer types, real types with floating-point, Boolean, character, string and object type. We will continue with the **variables**, with their characteristics, how to declare them, how they are assigned a value and what a variable initialization is. We will get familiar with the two major sets of data types in C# – **value types** and **reference types**. Finally we will examine different types of **literals** and their usage.

What Is a Variable?

A typical program uses various **values that change during its execution**. For example, we create a program that performs some calculations on the values entered by the user. The values entered by one user will obviously be different from those entered in by another user. This means that when creating the program, the programmer does not know what values will be introduced as input, and that makes it necessary to process all possible values a user may enter.

When a user enters a new value that will be used in the process of calculation, we can preserve it (temporarily) in the random access memory of our computer. The values in this part of memory change (vary) throughout execution and this has led to their name – **variables**.

Data Types

Data types are sets (ranges) of values that have similar characteristics. For instance **`byte`** type specifies the set of integers in the range of [0 … 255].

Characteristics

Data types are characterized by:

- **Name** – for example, **`int`**;
- **Size** (how much memory they use) – for example, 4 bytes;
- **Default value** – for example 0.

Types

Basic data types in C# are distributed into the following **types**:

- Integer types – **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**;
- Real floating-point types – **float**, **double**;
- Real type with decimal precision – **decimal**;
- Boolean type – **bool**;
- Character type – **char**;
- String – **string**;
- Object type – **object**.

These data types are called **primitive (built-in types)**, because they are embedded in C# language at the lowest level. The table below represents the above mentioned data types, their range and their default values:

Data Types	Default Value	Minimum Value	Maximum Value
sbyte	0	-128	127
byte	0	0	255
short	0	-32768	32767
ushort	0	0	65535
int	0	-2147483648	2147483647
uint	0u	0	4294967295
long	0L	-9223372036854775808	9223372036854775807
ulong	0u	0	18446744073709551615
float	0.0f	$\pm1.5\times10^{-45}$	$\pm3.4\times10^{38}$
double	0.0d	$\pm5.0\times10^{-324}$	$\pm1.7\times10^{308}$
decimal	0.0m	$\pm1.0\times10^{-28}$	$\pm7.9\times10^{28}$
bool	false	Two possible values: **true** and **false**	
char	'\u0000'	'\u0000'	'\uffff'
object	null	-	-
string	null	-	-

Correspondence between C# and .NET Types

Primitive data types in C# have a direct correspondence with the types of the common type system (CTS) in .NET Framework. For instance, **int** type in C# corresponds to **System.Int32** type in CTS and to **Integer type in** VB.NET language, while **long** type in C# corresponds to **System.Int64 type in** CTS and to **Long type** in VB.NET language. Due to the common types system (CTS) in .NET Framework there is compatibility between different programming languages (like for instance, C#, Managed C++, VB.NET and F#). For the same reason **int**, **Int32** and **System.Int32** types in C# are actually different aliases for one and the same data type – signed 32-bit integer.

Integer Types

Integer types represent integer numbers and are **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long** and **ulong**. Let's examine them one by one.

The **sbyte** type is an **8-bit signed integer**. This means that the number of possible values for it is 2^8, i.e. 256 values altogether, and they can be both, positive and negative. The minimum value that can be stored in **sbyte** is **SByte.MinValue** = -128 (-2^7), and the maximum value is **SByte.MaxValue** = 127 (2^7-1). The default value is the number 0.

The **byte** type is an **8-bit unsigned integer** type. It also has 256 different integer values (2^8) that can only be nonnegative. Its default value is the number 0. The minimal taken value is **Byte.MinValue** = 0, and the maximum is **Byte.MaxValue** = 255 (2^8-1).

The **short** type is a **16-bit signed integer**. Its minimal value is **Int16.MinValue** = -32768 (-2^{15}), and the maximum is **Int16.MaxValue** = 32767 ($2^{15}-1$). The default value for **short** type is the number 0.

The **ushort** type is **16-bit unsigned integer**. The minimum value that it can store is **UInt16.MinValue** = 0, and the minimum value is – **UInt16.MaxValue** = 65535 ($2^{16}-1$). Its default value is the number 0.

The next integer type that we will consider is **int**. It is a **32-bit signed integer**. As we can notice, the growth of bits increases the possible values that a type can store. The default value for **int** is 0. Its minimal value is **Int32.MinValue** = -2,147,483,648 (-2^{31}), and its maximum value is **Int32.MaxValue** = 2,147,483,647 ($2^{31}-1$).

The **int** type is **the most often used type in programming**. Usually programmers use **int** when they work with integers because this type is natural for the 32-bit microprocessor and is sufficiently "big" for most of the calculations performed in everyday life.

The **uint** type is **32-bit unsigned integer** type. Its default value is the number **0u** or **0U** (the two are equivalent). The '**u**' letter indicates that the number is of type **uint** (otherwise it is understood as **int**). The minimum

value that it can take is **UInt32.MinValue** = 0, and the maximum value is **UInt32.MaxValue** = 4,294,967,295 (2^{32}-1).

The **long** type is a **64-bit signed type** with a default value of **0l** or **0L** (the two are equivalent but it is preferable to use '**L**' because the letter '**l**' is easily mistaken for the digit one '**1**'). The '**L**' letter indicates that the number is of type **long** (otherwise it is understood **int**). The minimal value that can be stored in the **long** type is **Int64.MinValue** = -9,223,372,036,854,775,808 (-2^{63}) and its maximum value is **Int64.MaxValue** = 9,223,372,036,854, 775,807 (2^{63}-1).

The **biggest integer type** is the **ulong** type. It is a 64-bit unsigned type, which has as a default value – the number **0u**, or **0U** (the two are equivalent). The suffix '**u**' indicates that the number is of type **ulong** (otherwise it is understood as **long**). The minimum value that can be recorded in the **ulong** type is **UInt64.MinValue** = 0 and the maximum is **UInt64.MaxValue** = 18,446,744,073,709,551,615 (2^{64}-1).

Integer Types – Example

Consider an example in which we declare several variables of the integer types we know, we initialize them and print their values to the console:

```
// Declare some variables
byte centuries = 20;
ushort years = 2000;
uint days = 730480;
ulong hours = 17531520;
// Print the result on the console
Console.WriteLine(centuries + " centuries are " + years +
   " years, or " + days + " days, or " + hours + " hours.");

// Console output:
// 20 centuries are 2000 years, or 730480 days, or 17531520
// hours.

ulong maxIntValue = UInt64.MaxValue;
Console.WriteLine(maxIntValue); // 18446744073709551615
```

You would be able to see the declaration and initialization of a variable in detail in sections "Declaring Variables" and "Initialization of Variables" below, and it would become clear from the examples.

In the code snippet above, we demonstrate the use of integer types. For small numbers we use **byte** type, and for very large – **ulong**. We use unsigned types because all used values are positive numbers.

Real Floating-Point Types

Real types in C# are the real numbers we know from mathematics. They are represented by a **floating-point** according to the standard IEEE 754 and are **float** and **double**. Let's consider in details these two data types and understand what their similarities and differences are.

Real Type Float

The first type we will consider is the 32-bit real **floating-point type float**. It is also known as a **single precision real number**. Its default value is **0.0f** or **0.0F** (both are equivalent). The character '**f**' when put at the end explicitly indicates that the number is of type **float** (because by default all real numbers are considered **double**). More about this special suffix we can read bellow in the "Real Literals" section. The considered type has accuracy up to seven decimal places (the others are lost). For instance, if the number **0.123456789** is stored as type **float** it will be rounded to **0.1234568**. The range of values, which can be included in a float type (rounded with accuracy of 7 significant decimal digits), range from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$.

Special Values of the Real Types

The real data types have also several special values that are not real numbers but are mathematical abstractions:

- **Negative infinity -∞ (Single.NegativeInfinity)**. It is obtained when for instance we are dividing **-1.0f** by **0.0f**.
- **Positive infinity +∞ (Single.PositiveInfinity)**. It is obtained when for instance we are dividing **1.0f** by **0.0f**.
- **Uncertainty (Single.NaN)** – means that an invalid operation is performed on real numbers. It is obtained when for example we divide **0.0f** by **0.0f**, as well as when calculating square root of a negative number.

Real Type Double

The second real **floating-point type** in the C# language is the **double** type. It is also called **double precision real number** and is a 64-bit type with a default value of **0.0d** and **0.0D** (the suffix '**d**' is not mandatory because by default all real numbers in C# are of type **double**). This type has precision of 15 to 16 decimal digits. The range of values, which can be recorded in **double** (rounded with precision of 15-16 significant decimal digits), is from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$.

The **smallest real value** of type **double** is the constant **Double.MinValue** = **-1.79769e+308** and the largest is **Double.MaxValue** = **1.79769e+308**. The closest to 0 positive number of type **double** is **Double.Epsilon** = **4.94066e-324**. As with the type **float** the variables of type **double** can take the special

values: **Double.PositiveInfinity** (+∞), **Double.NegativeInfinity** (-∞) and **Double.NaN** (invalid number).

Real Floating-Point Types – Example

Here is an example in which we declare variables of real number types, assign values to them and print them:

```
float floatPI = 3.14f;
Console.WriteLine(floatPI); // 3.14
double doublePI = 3.14;
Console.WriteLine(doublePI); // 3.14
double nan = Double.NaN;
Console.WriteLine(nan); // NaN
double infinity = Double.PositiveInfinity;
Console.WriteLine(infinity); // Infinity
```

Precision of the Real Types

In mathematics the real numbers in a given range are countless (as opposed to the integers in that range) as between any two real numbers **a** and **b** there are countless other real numbers **c** where **a < c < b**. This requires real numbers to be stored in computer memory with a limited accuracy.

Since mathematics and physics mostly work with **extremely large numbers** (positive and negative) and with **extremely small numbers** (very close to zero), real types in computing and electronic devices must be stored and processed appropriately. For example, according to the physics the mass of electron is approximately $9.109389*10^{-31}$ kilograms and in 1 mole of substance there are approximately $6.02*10^{23}$ atoms. Both these values can be stored easily in **float** and **double** types.

Due to its flexibility, the modern **floating-point representation of real numbers** allows us to work with a maximum number of significant digits for very large numbers (for example, positive and negative numbers with hundreds of digits) and with numbers very close to zero (for example, positive and negative numbers with hundreds of zeros after the decimal point before the first significant digit).

Accuracy of Real Types – Example

The real types in C# we went over – **float** and **double** – differ not only by the range of possible values they can take, but also by their **precision** (the number of decimal digits, which they can preserve). The first type has a precision of 7 digits, the second – 15-16 digits.

Consider an example in which we declare several variables of the known real types, initialize them and print their values on the console. The purpose of the example is to illustrate the difference in their accuracy:

```
// Declare some variables
float floatPI = 3.141592653589793238f;
double doublePI = 3.141592653589793238;

// Print the results on the console
Console.WriteLine("Float PI is: " + floatPI);
Console.WriteLine("Double PI is: " + doublePI);

// Console output:
// Float PI is: 3.141593
// Double PI is: 3.14159265358979
```

We see that the number **π** which we declared as **float**, is rounded to the 7-th digit, and the one we declared **double** – to 15-th digit. We can conclude that the real type **double** retains much greater precision than **float**, thus if we need a greater precision after the decimal point, we will use it.

About the Presentation of the Real Types

Real floating-point numbers in C# consist of three components (according to the standard IEEE 754): **sign** (1 or -1), **mantissa** and **order** (**exponent**), and their values are calculated by a complex formula. More detailed information about the representation of the real numbers is provided in the chapter "Numeral Systems" where we will take an in-depth look at the representation of numbers and other data types in computing.

Errors in Calculations with Real Types

In calculations with real floating-point data types it is possible to observe **strange behavior**, because during the representation of a given real number it often happens **to lose accuracy**. The reason for this is the inability of some real numbers to be represented exactly as a sum of negative powers of the number 2. Examples of numbers that do not have an accurate representation in **float** and **double** types are for instance 0.1, 1/3, 2/7 and other. Here is a sample C# code, which demonstrates errors in calculations with floating-point numbers in C#:

```
float f = 0.1f;
Console.WriteLine(f); // 0.1 (correct due to rounding)
double d = 0.1f;
Console.WriteLine(d); // 0.100000001490116 (incorrect)

float ff = 1.0f / 3;
Console.WriteLine(ff); // 0.3333333 (correct due to rounding)
double dd = ff;
Console.WriteLine(dd); // 0.333333343267441 (incorrect)
```

The reason for the unexpected result in the first example is the fact that the number 0.1 (i.e. 1/10) **has no accurate representation** in the real floating-point number format IEEE 754 and its approximate value is recorded. When printed directly the result looks correct because of the rounding. The rounding is done during the conversion of the number to string in order to be printed on the console. When switching from **float** to **double** the approximate representation of the number in the IEEE 754 format is more noticeable. Therefore, the rounding does no longer hide the incorrect representation and we can observe the errors in it after the eighth digit.

In the second case the number **1/3 has no accurate representation** and is rounded to a number very close to 0.3333333. The value of this number is clearly visible when it is written in **double** type, which preserves more significant digits.

Both examples show that **floating-point number arithmetic can produce mistakes**, and is therefore not appropriate for precise financial calculations. Fortunately, C# supports decimal precision arithmetic where numbers like 0.1 are presented in the memory without rounding.

	Not all real numbers have accurate representation in float and double types. For example, the number 0.1 is representted rounded in float type as 0.099999994.

Real Types with Decimal Precision

C# supports the so-called **decimal floating-point arithmetic,** where numbers are represented via the decimal numeral system rather than the binary one. Thus, the decimal floating point-arithmetic type in C# **does not lose accuracy** when storing and processing floating-point numbers.

The type of data for real numbers with **decimal precision** in C# is the 128-bit type **decimal**. It has a precision from 28 to 29 decimal places. Its minimal value is -7.9×10^{28} and its maximum value is $+7.9 \times 10^{28}$. The default value is **0.0m** or **0.0M**. The 'm' character at the end indicates explicitly that the number is of type **decimal** (because by default all real numbers are of type **double**). The closest to **0** numbers, which can be recorded in decimal, are $\pm 1.0 \times 10^{-28}$. It is obvious that **decimal** can store neither very big positive or negative numbers (for example, with hundreds of digits), nor values very close to **0**. However, this type is almost perfect for financial calculations because it represents the numbers as a sum of powers of 10 and losses from rounding are much smaller than when using binary representation. The real numbers of type **decimal** are **extremely convenient for financial calculations** – calculation of revenues, duties, taxes, interests, payments, etc.

Here is an example in which we declare a variable of type **decimal** and assign its value:

```
decimal decimalPI = 3.14159265358979323846m;
Console.WriteLine(decimalPI); // 3.14159265358979323846
```

The number **decimalPI**, which we declared of type **decimal**, is not rounded even with a single position because we use it with **precision of 21 digits**, which fits in the type **decimal** without being rounded.

Because of the high precision and the **absence of anomalies** during calculations (which exist for **float** and **double**), the **decimal** type is extremely suitable for financial calculations where accuracy is critical.

Despite its smaller range, the decimal type retains precision for all decimal numbers it can store! This makes it much more suitable for precise calculations, and very appropriate for financial ones.

The main difference between **real floating-point numbers** and **real numbers with decimal precision** is the accuracy of calculations and the extent to which they round up the stored values. The **double** type allows us to work with very large values and values very close to zero but at the expense of accuracy and some unpleasant rounding errors. The **decimal** type has smaller range but ensures **greater accuracy** in computation, as well as absence of anomalies with the decimal numbers.

If you perform calculations with money use the decimal type instead of float or double. Otherwise, you may encounter unpleasant anomalies while calculating and errors as a result!

As all calculations with data of type **decimal** are done completely by software, rather than directly at a low microprocessor level, the calculations of this type are from **several tens to hundreds of times slower** than the same calculations with **double**, so use this type only when it is really necessary.

Boolean Type

Boolean type is declared with the keyword **bool**. It has two possible values: **true** and **false**. Its default value is **false**. It is used most often to store the calculation **result of logical expressions**.

Boolean Type – Example

Consider an example in which we declare several variables from the already known types, initialize them, compare them and print the result on the console:

```
// Declare some variables
```

```
int a = 1;
int b = 2;
// Which one is greater?
bool greaterAB = (a > b);
// Is 'a' equal to 1?
bool equalA1 = (a == 1);
// Print the results on the console
if (greaterAB)
{
  Console.WriteLine("A > B");
}
else
{
  Console.WriteLine("A <= B");
}

Console.WriteLine("greaterAB = " + greaterAB);
Console.WriteLine("equalA1 = " + equalA1);

// Console output:
// A <= B
// greaterAB = False
// equalA1 = True
```

In the example above, we declare two variables of type **int**, compare them and assign the result to the **bool** variable **greaterAB**. Similarly, we do the same for the variable **equalA1**. If the variable **greaterAB** is **true**, then **A > B** is printed on the console, otherwise **A <= B** is printed.

Character Type

Character type is a **single character** (16-bit number of a Unicode table character). It is declared in C# with the keyword **char**. **The Unicode table** is a technological standard that represents any character (letter, punctuation, etc.) from all human languages as writing systems (all languages and alphabets) with an integer or a sequence of integers. More about the **Unicode** table can be found in the chapter "Strings and Text Processing". The smallest possible value of a **char** variable is **0**, and the largest one is 65535. The values of type **char** are letters or other characters, and are enclosed in apostrophes.

Character Type – Example

Consider an example in which we declare one variable of type **char**, initialize it with value **'a'**, then **'b'**, then **'A'** and print the Unicode values of these letters to the console:

```
// Declare a variable
char ch = 'a';
// Print the results on the console
Console.WriteLine(
  "The code of '" + ch + "' is: " + (int)ch);
ch = 'b';
Console.WriteLine(
  "The code of '" + ch + "' is: " + (int)ch);
ch = 'A';
Console.WriteLine(
  "The code of '" + ch + "' is: " + (int)ch);

// Console output:
// The code of 'a' is: 97
// The code of 'b' is: 98
// The code of 'A' is: 65
```

Strings

Strings are sequences of characters. In C# they are declared by the keyword **string**. Their default value is **null**. Strings are enclosed in quotation marks. Various text-processing operations can be performed using strings: concatenation (joining one string with another), splitting by a given separator, searching, replacement of characters and others. More information about text processing can be found in the chapter "Strings and Text Processing", where you will find detailed explanation on what a string is, what its applications are and how we can use it.

Strings – Example

Consider an example in which we declare several variables of type **string**, initialize them and print their values on the console:

```
// Declare some variables
string firstName = "John";
string lastName = "Smith";
string fullName = firstName + " " + lastName;
// Print the results on the console
Console.WriteLine("Hello, " + firstName + "!");
Console.WriteLine("Your full name is " + fullName + ".");

// Console output:
// Hello, John!
// Your full name is John Smith.
```

Object Type

Object type is a special type, which is the parent of all other types in the .NET Framework. Declared with the keyword **object**, it can take values from **any other type**. It is a reference type, i.e. an index (address) of a memory area which stores the actual value.

Using Objects – Example

Consider an example in which we declare several variables of type **object**, initialize them and print their values on the console:

```
// Declare some variables
object container1 = 5;
object container2 = "Five";

// Print the results on the console
Console.WriteLine("The value of container1 is: " + container1);
Console.WriteLine("The value of container2 is: " + container2);

// Console output:
// The value of container1 is: 5
// The value of container2 is: Five.
```

As you can see from the example, we can store the value of any other type in an **object** type variable. This makes the **object** type a universal data container.

Nullable Types

Nullable types are specific **wrappers** around the value types (as **int**, **double** and **bool**) that allow storing data with a **null** value. This provides opportunity for types that generally do not allow lack of value (i.e. value **null**) to be used as reference types and to accept both normal values and the special one **null**. Thus nullable types **hold an optional value**.

Wrapping a given type as nullable can be done in two ways:

```
Nullable<int> i1 = null;
int? i2 = i1;
```

Both declarations are equivalent. The easiest way to perform this operation is to add a question mark (**?**) after the type, for example **int?**, the more difficult is to use the **Nullable<…>** syntax.

Nullable types are **reference types** i.e. they are reference to an object in the dynamic memory, which contains their actual value. They may or may not have a value and can be used as normal primitive data types, but with some specifics, which are illustrated in the following example:

```
int i = 5;
int? ni = i;
Console.WriteLine(ni); // 5

// i = ni; // this will fail to compile
Console.WriteLine(ni.HasValue); // True
i = ni.Value;
Console.WriteLine(i); // 5

ni = null;
Console.WriteLine(ni.HasValue); // False
//i = ni.Value; // System.InvalidOperationException
i = ni.GetValueOrDefault();
Console.WriteLine(i); // 0
```

The example above shows how a **nullable variable** (**int?**) can have a value directly added even if the value is non-nullable (**int**). The opposite is not directly possible. For this purpose, the nullable types' property **Value** can be used. It returns the value stored in the nullable type variable, or produces an error (**InvalidOperationException**) during program execution if the value is missing (**null**). In order to check whether a variable of nullable type has a value assigned, we can use the Boolean property **HasValue**. Another useful method is **GetValueOrDefault()**. If the nullable type variable has a value, this method will return its value, else it will return the default value for the nullable type (most commonly **0**).

Nullable types are used for storing information, which is **not mandatory**. For example, if we want to store data for a student such as the first name and last name as mandatory and age as not required, we can use type **int?** for the age variable:

```
string firstName = "John";
string lastName = "Smith";
int? age = null;
```

Variables

After reviewing the main data types in C# let's see how we can use them. In order to work with data we should use **variables**. We have already seen their usage in the examples, but now let's look at them in more detail.

A **variable** is a **container of information**, which can change its value. It provides means for:

- storing information;
- retrieving the stored information;

- modifying the stored information.

In C# programming, you will use variables to store and process information all the time.

Characteristics of Variables

Variables are characterized by:

- **name** (identifier), for example **age**;
- **type** (of the information preserved in them), for example **int**;
- **value** (stored information), for example **25**.

A **variable is a named area of memory**, which stores a value from a particular data type, and that area of memory is accessible in the program by its name. Variables can be stored directly in the operational memory of the program (in the stack) or in the dynamic memory in which larger objects are stored (such as character strings and arrays).

Primitive data types (numbers, **char**, **bool**) are called value types because they store their value directly in the program stack.

Reference data types (such as strings, objects and arrays) are an address, pointing to the dynamic memory where their value is stored. They can be dynamically allocated and released i.e. their size is not fixed in advance contrary to the case of value types.

More information about the value and reference data types is provided in the section "Value and Reference Types".

Naming Variables – Rules

When we want the compiler to allocate a memory area for some information which is used in our program we must provide a **name** for it. It works like an identifier and allows referring to the relevant memory area.

The name of the variable can be any of our choice but must follow certain rules defined in the C# language specification:

- Variable names can contain the letters **a**-**z**, **A**-**Z**, the digits **0**-**9** as well as the character '**_**'.
- Variable names cannot start with a digit.
- Variable names cannot coincide with a **keyword** of the C# language. For example, **base**, **char**, **default**, **int**, **object**, **this**, **null** and many others cannot be used as variable names.

A list of the C# keywords can be found in the section "Keywords" in chapter "Introduction to Programming". If we want to name a variable like a keyword, we can add a prefix to the name – "**@**". For example, **@char** and **@null** are valid variable names while **char** and **null** are invalid.

Naming Variables – Examples

Proper names:

- **name**
- **first_Name**
- **_name1**

Improper names (will lead to compilation error):

- **1** (digit)
- **if** (keyword)
- **1name** (starts with a digit)

Naming Variables – Recommendations

We will provide some recommendations how to name your variables, since not all names, allowed by the compiler, are appropriate for the variables.

- The names should be descriptive and explain what the variable is used for. For example, an appropriate name for a variable storing a person's name is **personName** and inappropriate name is **a37**.
- Only **Latin characters** should be used. Although Cyrillic is allowed by the compiler, it is not a good practice to use it in variable names or in the rest of the identifiers within the program.
- In C# it is generally accepted that variable names should **start with a small letter** and include small letters, every new word, however, starts with a capital letter. For instance, the name **firstName** is correct and better to use than **firstname** or **first_name**. Usage of the character **_** in the variable names is considered a bad naming style.
- Variable names should be **neither too long nor too short** – they just need to clarify the purpose of the variable within its context.
- Uppercase and lowercase letters should be used carefully as C# distinguishes them. For instance, **age** and **Age** are different variables.

Here are some examples of well-named variables:

- **firstName**
- **age**
- **startIndex**
- **lastNegativeNumberIndex**

And here are some examples for poorly named variables (although the names are correct from the C# compiler's perspective):

- **_firstName** (starts with **_**)

- **last_name** (contains _)
- **AGE** (is written with capital letters)
- **Start_Index** (starts with capital letter and contains _)
- **lastNegativeNumber_Index** (contains _)
- **a37** (the name is not descriptive and does not clearly provide the purpose of the variable)
- **fullName23**, **fullName24**, etc. (it is not appropriate for a variable name to contain digits unless this improves the clarity of the variable used; if you need to have multiple variables with similar names ending in a different number, storing the same or similar type of data, it may be more appropriate to create a single collection or array variable and name it **fullNamesList**, for example).

Variables should have names, which **briefly explain their purpose**. When a variable is named with an inappropriate name, it makes the program very difficult to read and modify later (after a while, when we have forgotten how it works). For further explanation on the proper naming of variables refer to chapter "High-Quality Programming Code".

	Always try to use short and precise names when naming the variables. Follow the rule that the variable name should state what it is used for, e.g. the name should answer the question "what value is stored in this variable". When this condition is not fulfilled then try to find a better name. Digits are not appropriate to be used in variable names.

Declaring Variables

When you declare a variable, you perform the following steps:

- specify its **type** (such as **int**);
- specify its **name** (identifier, such as **age**);
- optionally specify **initial value** (such as **25**) but this is not obligatory.

The **syntax for declaring variables** in C# is as follows:

```
<data type> <identifier> [= <initialization>];
```

Here is an **example** of declaring variables:

```
string name;
int age;
```

Assigning a Value

Assigning a value to a variable is the act of providing a value that must be stored in the variable. This operation is performed by the assignment operator "**=**". On the left side of the operator we put the variable name and on the right side – its new value.

Here is an example of assigning values to variables:

```
name = "John Smith";
age = 25;
```

Initialization of Variables

The word **initialization** in programming means specifying an initial value. When setting value to variables at the time of their declaration we actually initialize them.

Default Variable Values

Each data type in C# has a **default value** (default initialization) which is used when there is no explicitly set value for a given variable. We can use the following table to see the default values of the types, which we already got familiar with:

Data Type	Default Value
sbyte	0
byte	0
short	0
ushort	0
int	0
uint	0u
long	0L
ulong	0u

Data Type	Default Value
float	0.0f
double	0.0d
decimal	0.0m
bool	false
char	'\u0000'
string	null
object	null

Let's summarize how to declare variables, initialize them and assign values to them with the following example:

```
// Declare and initialize some variables
byte centuries = 20;
ushort years = 2000;
decimal decimalPI = 3.141592653589793238m;
bool isEmpty = true;
char ch = 'a';
```

```
string firstName = "John";

ch = (char)5;
char secondChar;

// Here we use an already initialized variable and reassign it
secondChar = ch;
```

Value and Reference Types

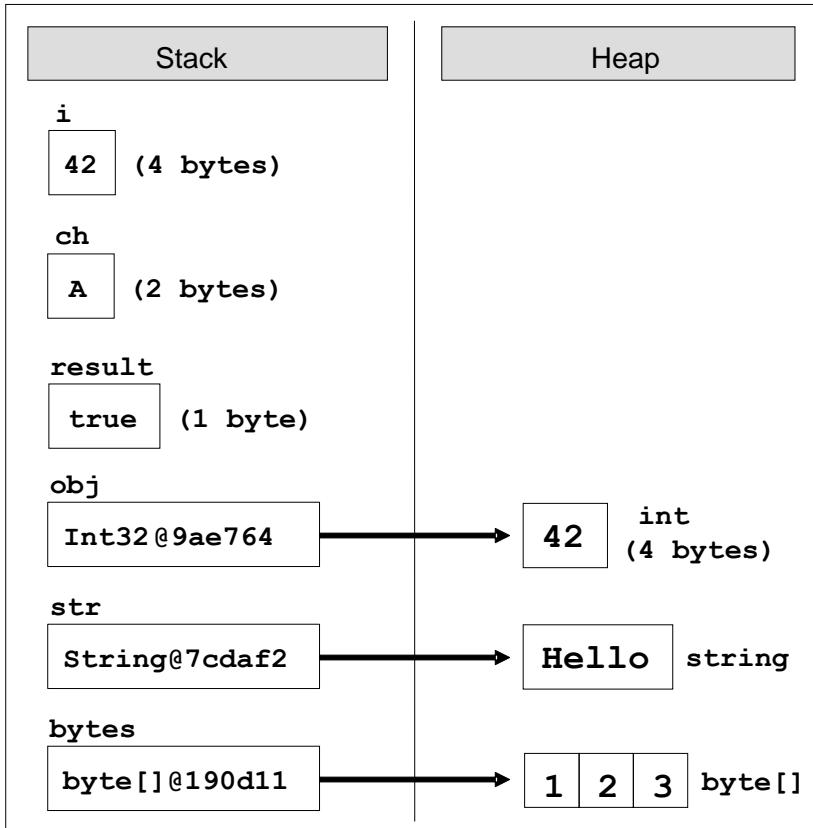
Data types in C# are two types: **value** and **reference**.

Value types are stored in the program execution stack and directly contain their value. Value types are the primitive numeric types, the character type and the Boolean type: **sbyte**, **byte**, **short**, **ushort**, **int**, **long**, **ulong**, **float**, **double**, **decimal**, **char**, **bool**. The memory allocated for them is released when the program exits their range, i.e. when the block of code in which they are defined completes its execution. For example, a variable declared in the method **Main()** of the program is stored in the stack until the program completes execution of this method, i.e. until it finishes (C# programs terminate after fully executing the **Main()** method).

Reference types keep a **reference** (**address**), in the program execution stack, and that reference points to the **dynamic memory** (**heap**), where their value is stored. The reference is a **pointer** (address of the memory cell) indicating the actual location of the value in the heap. An example of a value at address in the stack for execution is **0x00AD4934**. The reference has a type. The reference can only point to objects of the same type, i.e. it is a strongly typed pointer. All reference types can hold a **null** value. This is a special service value, which means that there is no value.

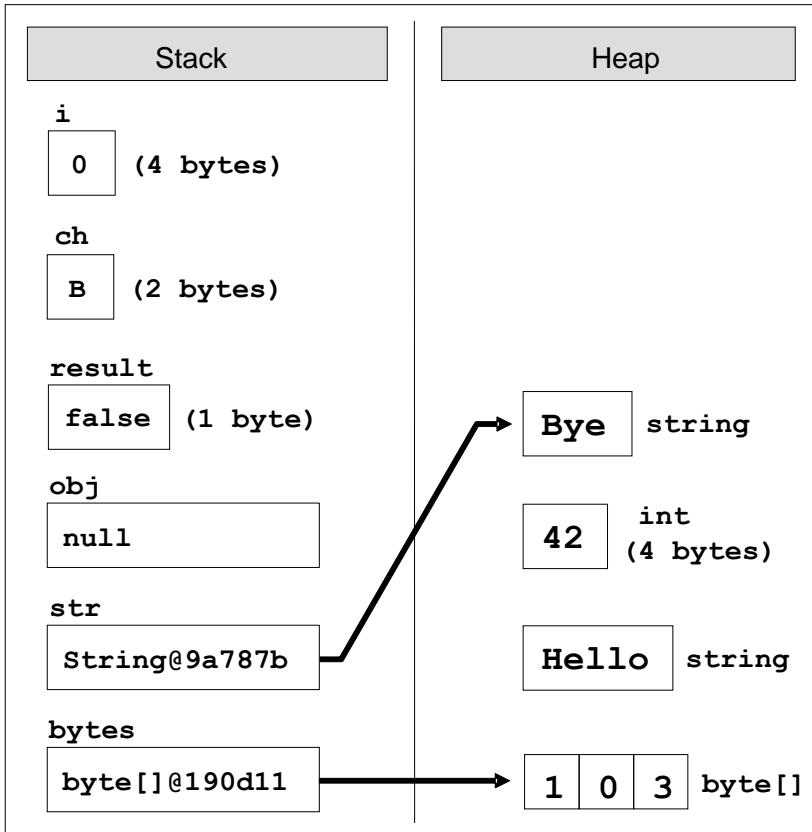
Reference types allocate **dynamic memory** for their creation. They also release some dynamic memory for a **memory cleaning (garbage collector),** when it is no longer used by the program. It is unknown exactly when a given reference variable will be released of the garbage collector as this depends on the memory load and other factors. Since the allocation and release of memory is a slow operation, it can be said that the reference types are slower than the value ones.

As **reference data types** are **allocated and released dynamically** during program execution, their size might not be known in advance. For example, a variable of type **string** can contain text data which varies in length. Actually the **string** text value is stored in the dynamic memory and can occupy a different volume (count of bytes) while the **string** variable stores the address of the text value.

Reference types are all **classes**, **arrays** and **interfaces** such as the types: **object**, **string**, **byte[]**. We will learn about classes, objects, strings, arrays and interfaces in the next chapters of this book. For now, it is enough to know

that all types, which are not value, are reference and their values are stored in the **heap** (the dynamically allocated memory).

Value and Reference Types and the Memory

In this example we will illustrate how value and reference types are **represented in memory**. Consider the execution of the following programming code:

```
int i = 42;
char ch = 'A';
bool result = true;
object obj = 42;
string str = "Hello";
byte[] bytes = { 1, 2, 3 };
```

At this point the variables are located in the memory as follows:

If we now execute the following code, which changes the values of the variables, we will see **what happens to the memory** when changing the value and reference types:

```
i = 0;
ch = 'B';
result = false;
obj = null;
str = "Bye";
bytes[1] = 0;
```

After these changes the variables and their values are **located in the memory** as follows:

As you can see from the figure, a change in a value type (**i = 0**) changes its value **directly into the stack**. When changing a reference type, things are different: the value is **changed in the heap** (**bytes[1] = 0**). The variable that keeps the array reference remains unchanged (**0x00190D11**). When assigning a **null** value in a reference type, that reference is disconnected from its value and the variable remains with no value (**obj = null**).

When assigning new value to an object (a reference type variable) the new object is allocated in the heap (the dynamic memory) while the old object remains free (unreferenced). The reference is redirected to the new object (**str = "Bye"**) while the old objects (**"Hello"**) will be cleaned at some moment

by the **garbage collector** (the .NET Framework’s internal system for automatic memory cleaning) as they are not in use anymore.

Literals

Primitive types, which we already met, are special data types built into the C# language. Their values specified in the source code of the program are called **literals**. One example will make this clearer:

```
bool result = true;
char capitalC = 'C';
byte b = 100;
short s = 20000;
int i = 300000;
```

In the above example, literals are **true**, **'C'**, **100**, **20000** and **300000**. They are variable values set directly in the source code of the program.

Types of Literals

In C# language, there are several types of literals:

- Boolean
- Integer
- Real
- Character
- String
- Object literal **null**

Boolean Literals

Boolean literals are:

- **true**
- **false**

When we assign a value to a variable of type **bool** we can use only one of these two values or a Boolean expression (which is calculated to **true** or **false**).

Boolean Literals – Example

Here is an example of a declaration of a variable of type **bool** and assigning a value, which represents the Boolean literal **true**:

```
bool result = true;
```

Integer Literals

Integer literals are **sequences of digits**, a sign (**+**, **-**), suffixes and prefixes. Using prefixes we can present integers in the program source in decimal or hexadecimal format. More information about the different numeral systems we can find in the chapter "Numeral Systems". In the integer literals the following prefixes and suffixes may take part:

- "**0x**" and "**0X**" as prefix indicates hexadecimal values, for example **0xA8F1**;
- '**l**' and '**L**' as suffix indicates **long** type data, for example **357L**.
- '**u**' and '**U**' as suffix indicates **uint** or **ulong** data type, for example **112u**.

By default (if no suffix is used) the integer literals are of type **int**.

Integer Literals – Examples

Here are some examples of using integer literals:

```
// The following variables are initialized with the same value
int numberInDec = 16;
int numberInHex = 0x10;

// This will cause an error, because the value 234L is not int
int longInt = 234L;
```

Real Literals

Real literals are a **sequence of digits**, a sign (**+**, **-**), suffixes and the decimal point character. We use them for values of type **float**, **double** and **decimal**. Real literals can be represented in exponential format. They also use the following indications:

- '**f**' and '**F**' as suffixes mean data of type **float**;
- '**d**' and '**D**' as suffixes mean data of type **double**;
- '**m**' and '**m**' as suffixes mean data of type **decimal**;
- '**e**' is an exponent, for example, "**e-5**" means the integer part multiplied by 10^{-5}.

By default (if there is no suffix), the real numbers are of type **double**.

Real Literals – Examples

Here are some examples of real literals' usage:

```
// The following is the correct way of assigning a value:
float realNumber = 12.5f;
```

```
// This is the same value in exponential format:
realNumber = 1.25e+1f;

// The following causes an error, because 12.5 is double
float realNumber = 12.5;
```

Character Literals

Character literals are **single characters enclosed in apostrophes** (single quotes). We use them to set the values of type **char**. The value of a character literal can be:

- a character, for example **'A'**;
- a character code, for example **'\u0065'**;
- an escaping sequence;

Escaping Sequences

Sometimes it is necessary to work with characters that are not displayed on the keyboard or with characters that have special meanings, such as the "new line" character. They **cannot be represented directly** in the source code of the program and in order to use them we need special techniques, which we will discuss now.

Escaping sequences are literals. They are a sequence of special characters, which describe a character that cannot be written directly in the source code. This is, for instance, the "**new line**" character.

There are many examples of characters that cannot be represented directly in the source code: a double quotation mark, tab, new line, backslash and others. Here are some of **the most frequently used escaping sequences**:

- **\'** – single quote
- **\"** – double quotes
- **** – backslash
- **\n** – new line
- **\t** – offset (tab)
- **\uXXXX** – char specified by its Unicode number, for example **\u03A7**.

The character **** (backslash) is also called an **escaping character** because it allows the display on screen (or other output device) of characters that have special meaning or effect and cannot be represented directly in the source code.

Escaping Sequences – Examples

Here are some examples of character literals:

```
// An ordinary character
char character = 'a';
Console.WriteLine(character);

// Unicode character code in a hexadecimal format
character = '\u003A';
Console.WriteLine(character);

// Assigning the single quotiation character (escaped as \')
character = '\'';
Console.WriteLine(character);

// Assigning the backslash character (escaped as \\)
character = '\\';
Console.WriteLine(character);

// Console output:
// a
// :
// '
// \
```

String Literals

String literals are used for data of type **string**. They are a sequence of characters enclosed in double quotation marks.

All the escaping rules for the **char** type discussed above are also valid for **string** literals.

Strings can be preceded by the **@** character that specifies a **quoted string** (**verbatim string**). In quoted strings the rules for escaping are not valid, i.e. the character **** means **** and is not an escaping character. Only one character needs to be escaped in the quoted strings – the character **"** (double-quotes) and it is escaped in the following way – by repeating it **""** (double double-quotes). All other characters are treated literally, even the new line. Quoted strings are often used for the file system paths naming.

String Literals – Examples

Here are few examples for string literals usage:

```
string quotation = "\"Hello, Jude\", he said.";
```

```
Console.WriteLine(quotation);
string path = "C:\\Windows\\Notepad.exe";
Console.WriteLine(path);
string verbatim = @"The \ is not escaped as \\.
I am at a new line.";
Console.WriteLine(verbatim);
// Console output:
// "Hello, Jude", he said.
// C:\Windows\Notepad.exe
// The \ is not escaped as \\.
// I am at a new line.
```

More about strings we will find in the chapter "Strings and Text Processing".

Exercises

1. **Declare several variables** by selecting for each one of them the most appropriate of the types **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long** and **ulong** in order to assign them the following values: 52,130; -115; 4825932; 97; -10000; 20000; 224; 970,700,000; 112; -44; -1,000,000; 1990; 123456789123456789.

2. Which of the following values can be assigned to variables of type **float**, **double** and **decimal**: 5, -5.01, 34.567839023; 12.345; 8923.1234857; 3456.091124875956542151256683467?

3. Write a program, which **compares correctly two real numbers** with accuracy at least **0.000001**.

4. **Initialize** a variable of type **int** with a value of 256 in **hexadecimal** format (256 is 100 in a numeral system with base 16).

5. Declare a variable of type **char** and assign it as a value the character, which has **Unicode** code, 72 (use the Windows calculator in order to find hexadecimal representation of 72).

6. Declare a variable **isMale** of type **bool** and assign a value to it depending on your gender.

7. Declare two variables of type **string** with values "Hello" and "World". Declare a variable of type **object**. Assign the value obtained of concatenation of the two string variables (add space if necessary) to this variable. Print the variable of type **object**.

8. Declare two variables of type **string** and give them values "Hello" and "World". Assign the value obtained by the concatenation of the two variables of type **string** (do not miss the space in the middle) to a variable of type **object**. Declare a third variable of type **string** and initialize it with the value of the variable of type **object** (you should use type casting).

9. Declare two variables of type **string** and assign them a value "**The "use" of quotations causes difficulties.**" (without the outer quotes). In one of the variables use quoted string and in the other do not use it.

10. Write a program to print a figure in the shape of a **heart** by the sign "**o**".

11. Write a program that prints on the console **isosceles triangle** which sides consist of the copyright character "**©**".

12. A company dealing with marketing wants to keep a data record of its **employees**. Each record should have the following characteristic – first name, last name, age, gender ('m' or 'f') and unique employee number (27560000 to 27569999). **Declare appropriate variables** needed to maintain the information for an employee by using the appropriate data types and attribute names.

13. Declare two variables of type **int**. Assign to them values 5 and 10 respectively. **Exchange (swap) their values** and print them.

Solutions and Guidelines

1. Look at the ranges of the numerical types in C# described in this chapter.

2. Consider the number of digits after the decimal point. Refer to the table that describes the sizes of the types **float**, **double** and **decimal**.

3. Two floating-point variables are considered equal if their difference is less than some predefined precision (e.g. **0.000001**):

```
bool equal = Math.Abs(a - b) < 0.000001;
```

4. Look at the section about Integer Literals. To easily convert numbers to a different numeral system use the built-in Windows calculator. For a **hexadecimal** representation of the literal **use prefix 0x**.

5. Look at the section about Character Literals.

6. Look at the section about Boolean Literals.

7. Look at the sections about Strings and Object Data Type.

8. Look at the sections about Strings and Object Data Type. To cast from object to string use **typecasting**:

```
string str = (string)obj;
```

9. Look at the section about Character Literals. It is necessary to use the **escaping character \"** or **verbatim strings**.

10. Use **Console.WriteLine(…)**, the character '**o**' and spaces.

11. Use **Console.WriteLine(…)**, the character **©** and spaces. Use Windows Character Map in order to find the Unicode code of the sign "**©**". Note that the console may display "**c**" instead of "**©**" if it does not

support Unicode. If this happens, you might be unable to do anything to fix it. Some versions of Windows just do not support Unicode in the console even when you explicitly set the character encoding to UTF-8:

```
Console.OutputEncoding = System.Text.Encoding.UTF8;
```

You may need to change the font of your console to some font that supports the "©" symbol, e.g. "**Consolas**" or "**Lucida Console**".

12. For the names use type **string**, for the gender use type **char** (only one char **m**/**f**), and for the unique number and age use some integer type.

13. Use **third temporary variable** for exchanging the variables:

```
int a = 5;
int b = 10;

int oldA = a;
a = b;
b = oldA;
```

To swap integer variables **other solutions** exist which do not use a third variable. For example, if we have two integer variables **a** and **b**:

```
int a = 5;
int b = 10;

a = a + b;
b = a - b;
a = a - b;
```

You might also use the **XOR swap algorithm** for exchanging integer values: http://en.wikipedia.org/wiki/XOR_swap_algorithm.

Chapter 3. Operators and Expressions

In This Chapter

In this chapter we will get acquainted with the **operators in C#** and **the actions they can perform** when used with the different data types. In the beginning, we will explain which operators have higher priority and we will analyze the different types of operators, according to the number of the arguments they can take and the actions they perform. In the second part, we will examine **the conversion of data types**. We will explain when and why it is needed to be done and how to work with different data types. At the end of the chapter, we will pay special attention to the **expressions** and how we should work with them. Finally, we have prepared exercises to strengthen our knowledge of the material in this chapter.

Operators

Every programming language uses **operators**, through which we can perform different actions on the data. Let's take a look at the operators in C# and see what they are for and how they are used.

What Is an Operator?

After we have learned how to declare and set a variable in the previous chapter, we will discuss how to perform various operations with them. For this purpose we will get familiar with operators.

Operators allow processing of primitive data types and objects. They take as an input one or more operands and return some value as a result. Operators in C# are special characters (such as "**+**", "**.**", "**^**", etc.) and they perform transformations on one, two or three operands. Examples of operators in C# are the signs for adding, subtracting, multiplication and division from math (**+**, **-**, *****, **/**) and the operations they perform on the integers and the real numbers.

Operators in C#

Operators in C# can be separated in several different categories:

- **Arithmetic** operators – they are used to perform simple mathematical operations.

- **Assignment** operators – allow assigning values to variables.
- **Comparison** operators – allow comparison of two literals and/or variables.
- **Logical** operators – operators that work with Boolean data types and Boolean expressions.
- **Binary** operators – used to perform operations on the binary representation of numerical data.
- **Type conversion** operators – allow conversion of data from one type to another.

Operator Categories

Below is a list of the operators, separated into categories:

<table>
<tr><th>Category</th><th>Operators</th></tr>
<tr><td>arithmetic</td><td>-, +, *, /, %, ++, --</td></tr>
<tr><td>logical</td><td>&&, ||, !, ^</td></tr>
<tr><td>binary</td><td>&, |, ^, ~, <<, >></td></tr>
<tr><td>comparison</td><td>==,!=, >, <, >=, <=</td></tr>
<tr><td>assignment</td><td>=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=</td></tr>
<tr><td>string concatenation</td><td>+</td></tr>
<tr><td>type conversion</td><td>(type), as, is, typeof, sizeof</td></tr>
<tr><td>other</td><td>., new, (), [], ?:, ??</td></tr>
</table>

Types of Operators by Number of Arguments

Operators can be separated into different types according to the number of arguments they could take:

Operator type	Number of arguments (operands)
unary	takes one operand
binary	takes two operands
ternary	takes three operands

All **binary operators** in C# are **left-associative**, i.e. the expressions are calculated from left to right, except for the assignment operators. All assignment operators and conditional operators **?:** and **??** are right-associative, i.e. the expressions are calculated from right to left. The unary operators are not associative.

Some of the operators in C# perform different operations on the different data types. For example the operator **+**. When it is used on numeric data

types (**int**, **long**, **float**, etc.), the operator performs mathematical addition. However, when we use it on strings, the operator concatenates (joins together) the content of the two variables/literals and returns the new string.

Operators – Example

Here is an example of using operators:

```
int a = 7 + 9;
Console.WriteLine(a); // 16

string firstName = "John";
string lastName = "Doe";

// Do not forget the space between them
string fullName = firstName + " " + lastName;
Console.WriteLine(fullName); // John Doe
```

The example shows how, as explained above, when the operator **+** is used on numbers it returns a numerical value, and when it is used on strings it returns concatenated strings.

Operator Precedence in C#

Some operators have **precedence** (priority) over others. For example, in math multiplication has precedence over addition. The operators with a higher precedence are calculated before those with lower. The operator **()** is used to **change the precedence** and like in math, it is calculated first.

The following table illustrates the precedence of the operators in C#:

Priority	Operators
Highest priority	**(**, **)**
	++, **--** (as postfix), **new**, **(type)**, **typeof**, **sizeof**
	++, **--** (as prefix), **+**, **-** (unary), **!**, **~**
	*****, **/**, **%**
	+ (string concatenation)
	+, **-**
…	**<<**, **>>**
	<, **>**, **<=**, **>=**, **is**, **as**
	==, **!=**
	&, **^**, **|**

Lowest priority	**&&**
	||
	?:, ??
	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

The operators located **upper in the table have higher precedence** than those below them, and respectively they have an advantage in the calculation of an expression. To change the precedence of an operator we can use brackets.

When we write expressions that are more complex or have many operators, it is recommended to **use brackets** to avoid difficulties in reading and understanding the code. For example:

```
// Ambiguous
x + y / 100

// Unambiguous, recommended
x + (y / 100)
```

Arithmetical Operators

The arithmetical operators in C# **+**, **-**, ***** are the same like the ones in math. They perform addition, subtraction and multiplication on numerical values and the result is also a numerical value.

The **division operator /** has different effect on integer and real numbers. When we divide an integer by an integer (like **int**, **long** and **sbyte**) the returned value is an integer (no rounding, the fractional part is cut). Such division is called an **integer division**. Example of integer division: 7 **/** 3 = 2.

Integer division by 0 is not allowed and causes a runtime exception **DivideByZeroException**. The remainder of integer division of integers can be obtained by the operator **%**. For example, 7 **%** 3 = 1, and –10 **%** 2 = 0.

When dividing two real numbers or two numbers, one of which is real (e.g. **float**, **double**, etc.), a **real division** is done (not integer), and the result is a real number with a whole and a fractional part. For example: 5.0 / 2 = 2.5. In the division of real numbers it is **allowed to divide by 0.0** and respectively the result is **+∞** (**Infinity**), **-∞** (**-Infinity**) or **NaN** (invalid value).

The operator for **increasing by one** (increment) **++** adds one unit to the value of the variable, respectively the operator **--** (**decrement**) subtracts one unit from the value. When we use the operators **++** and **--** as a **prefix** (when we place them immediately before the variable), the new value is calculated first and then the result is returned. When we use the same operators as **post-fix** (meaning when we place them immediately after the variable) the

original value of the operand is returned first, then the addition or subtraction is performed.

Arithmetical Operators – Example

Here are some examples of arithmetic operators and their effect:

```
int squarePerimeter = 17;
double squareSide = squarePerimeter / 4.0;
double squareArea = squareSide * squareSide;
Console.WriteLine(squareSide); // 4.25
Console.WriteLine(squareArea); // 18.0625

int a = 5;
int b = 4;
Console.WriteLine(a + b);      // 9
Console.WriteLine(a + (b++));  // 9
Console.WriteLine(a + b);      // 10
Console.WriteLine(a + (++b));  // 11
Console.WriteLine(a + b);      // 11
Console.WriteLine(14 / a);     // 2
Console.WriteLine(14 % a);     // 4

int one = 1;
int zero = 0;
// Console.WriteLine(one / zero); // DivideByZeroException

double dMinusOne = -1.0;
double dZero = 0.0;
Console.WriteLine(dMinusOne / zero); // -Infinity
Console.WriteLine(one / dZero); // Infinity
```

Logical Operators

Logical (Boolean) operators take Boolean values and return a Boolean result (true or **false**). The basic Boolean operators are "**AND**" (&&), "**OR**" (||), "**exclusive OR**" (^) and **logical negation** (!).

The following table contains the logical operators in C# and the operations that they perform:

x	y	!x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true

false	false	true	false	false	false

The table and the following example show that the logical "AND" (**&&**) returns true only when both variables contain truth. Logical "OR" (**||**) returns true when at least one of the operands is true. The logical negation operator (**!**) changes the value of the argument. For example, if the operand has a value **true** and a negation operator is applied, the new value will be **false**. The negation operator is a unary operator and it is placed before the argument. Exclusive "OR" (**^**) returns **true** if only one of the two operands has the value **true**. If the two operands have different values, exclusive "OR" will return the result **true**, if they have the same values it will return **false**.

Logical Operators – Example

The following example illustrates the usage of the logical operators and their actions:

```
bool a = true;
bool b = false;
Console.WriteLine(a && b);                // False
Console.WriteLine(a || b);                // True
Console.WriteLine(!b);                    // True
Console.WriteLine(b || true);             // True
Console.WriteLine((5 > 7) ^ (a == b));    // False
```

Laws of De Morgan

Logical operations fall under the **laws of De Morgan** from the mathematical logic:

```
!(a && b) == (!a || !b)
!(a || b) == (!a && !b)
```

The first law states that the negation of the conjunction (logical AND) of two propositions is equal to the disjunction (logical OR) of their negations.

The second law states that the negation of the disjunction of both statements is equivalent to the conjunction of their negations.

Operator for Concatenation of Strings

The operator **+** is used to **join strings** (**string**). It concatenates (joins) two or more strings and returns the result as a new string. If at least one of the arguments in the expression is of type **string**, and there are other operands of type different from **string**, they will be automatically converted to type **string**, which allows successful string concatenation.

It is fantastic how .NET runtime handles such operation incompatibilities for us on the fly, saving us some coding time and allowing us to concentrate on

the main objectives of our programming task! However, it is a good practice to not miss to cast the variables on which we wish to apply an operation; we should instead have them converted to the appropriate type for each operation, so that we are in full control of the end result and prevent implicit type casts. We will provide more detailed information on casting operations further down in the section "Type Conversion" of this chapter.

Operator for Concatenation of Strings – Example

Here is an example, which shows concatenations of two strings and a string with a number:

```
string csharp = "C#";
string dotnet = ".NET";
string csharpDotNet = csharp + dotnet;
Console.WriteLine(csharpDotNet); // C#.NET
string csharpDotNet4 = csharpDotNet + " " + 5;
Console.WriteLine(csharpDotNet4); // C#.NET 5
```

In the example we initialize two variables of type **string** and assign them values. On the third and fourth row we concatenate both strings and pass the results to the method **Console.WriteLine()** to print it on the console. On the next line we join the resulting string with a space and the number 5. We assign the returned value to the variable **csharpDotNet5**, which will automatically be converted to type **string**. On the last row we print the result.

Concatenation (joining, gluing) of strings is a slow operation and should be used carefully. It is recommended to use the StringBuilder class for iterative (repetitive) operations on strings.

In the chapter "Strings" we will explain in detail why the **StringBuilder class** must be used for join operations on strings performed in a loop.

Bitwise Operators

A bitwise operator is an operator that acts on the binary representation of numeric types. In computers all the data and particularly numerical data is represented as a series of ones and zeros. The **binary numeral system** is used for this purpose. For example, number **55** in the binary numeral system is represented as **00110111**.

Binary representation of data is convenient because zero and one in electronics can be implemented by Boolean circuits, in which zero is represented as "no electricity" or for example with a voltage of -5V and the one is presented as "have electricity" or say with voltage +5V.

We will examine in depth the **binary numeral system** in the chapter "Numeral Systems", but just for now we can consider that the numbers in computers are represented as ones and zeros, and bitwise operators are used to analyze and change those ones to zeros and vice versa.

Bitwise operators are very similar to the logical ones. In fact, we can imagine that the logical and bitwise operators perform the same thing but using different data types. Logical operators work with the values **true** and **false** (Boolean values), while bitwise operators work with numerical values and are applied bitwise over their binary representation, i.e., they work with the bits of the number (the digits **0** and **1** of which it consists). Just like the logical operators in C#, there are bitwise operators "AND" (**&**), bitwise "OR" (**|**), bitwise negation (**~**) and excluding "OR" (^).

Bitwise Operators and Their Performance

The bitwise operators' performance on binary digits **0** and **1** is shown in the following table:

x	y	~x	x & y	x | y	x ^ y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

As we see bitwise and logical operators are very much alike. The difference in the writing of "AND" and "OR" is that the logical operators are written with double ampersand (**&&**) and double vertical bar (**||**), and the bitwise – with a single ampersand or vertical bar (**&** and **|**). Bitwise and logical operators for exclusive "OR" are the same "^". For logical negation we use "**!**", while for bitwise negation (inversion) the "**~**" operator is used.

In programming there are two bitwise operators that have no analogue in logical operators. These are the **bit shift left** (**<<**) and **bit shift right** (**>>**). Used on numerical values, they move all the bits of the value to the left or right. The bits that fall outside the number are lost and replaced with 0.

The **bit shifting operators** are used in the following way: on the left side of the operator we place the variable (operand) with which we want to use the operator, on the right side we put a numerical value, indicating how many bits we want to offset. For example, **3 << 2** means that we want to move the bits of the number three, twice to the left. The number 3 presented in bits looks like this: "**0000 0011**". When you move twice left, the binary value will look like this: "**0000 1100**", and this sequence of bits is the number 12. If we look at the example we can see that actually we have multiplied the number by 4. Bit shifting itself can be represented as multiplication (bitwise shifting left) or division (bitwise shifting right) by a power of 2. This occurrence is due to the

nature of the binary numeral system. Example of moving to the right is **6 >> 2**, which means to move the binary number "**0000 0110**" with two positions to the right. This means that we will lose two right-most digits and feed them with zeros on the left. The end result will be "**0000 0001**" which is 1.

Bitwise Operators – Example

Here is an example of using bitwise operators. The binary representation of the numbers and the results of the bitwise operators are shown in the comments (green text):

```
byte a = 3;                 // 0000 0011 = 3
byte b = 5;                 // 0000 0101 = 5

Console.WriteLine(a | b);   // 0000 0111 = 7
Console.WriteLine(a & b);   // 0000 0001 = 1
Console.WriteLine(a ^ b);   // 0000 0110 = 6
Console.WriteLine(~a & b);  // 0000 0100 = 4
Console.WriteLine(a << 1);  // 0000 0110 = 6
Console.WriteLine(a << 2);  // 0000 1100 = 12
Console.WriteLine(a >> 1);  // 0000 0001 = 1
```

In the example we first create and initialize the values of two variables **a** and **b**. Then we print on the console the results of some bitwise operations on the two variables. The first operation that we apply is "OR". The example shows that for all positions where there was 1 in the binary representation of the variables **a** and **b**, there is also 1 in the result. The second operation is "AND". The result of the operation contains 1 only in the right-most bit, because the only place where **a** and **b** have 1 at the same time is their right-most bit. Exclusive "OR" returns ones only in positions where **a** and **b** have different values in their binary bits. Finally, the logical negation and bitwise shifting: left and right, are illustrated.

Comparison Operators

Comparison operators in C# are used to compare two or more operands. C# supports the following comparison operators:

- greater than (**>**)
- less than (**<**)
- greater than or equal to (**>=**)
- less than or equal to (**<=**)
- equality (**==**)
- difference (**!=**)

All comparison operators in C# are **binary** (take two operands) and the returned result is a Boolean value (**true** or **false**). Comparison operators have lower priority than arithmetical operators but higher than the assignment operators.

Comparison Operators – Example

The following example demonstrates the usage of comparison operators in C#:

```
int x = 10, y = 5;
Console.WriteLine("x > y : " + (x > y));    // True
Console.WriteLine("x < y : " + (x < y));    // False
Console.WriteLine("x >= y : " + (x >= y));  // True
Console.WriteLine("x <= y : " + (x <= y));  // False
Console.WriteLine("x == y : " + (x == y));  // False
Console.WriteLine("x != y : " + (x != y));  // True
```

In the example, first we create two variables **x** and **y** and we assign them the values 10 and 5. On the next line we print on the console using the method **Console.WriteLine(…)** the result from comparing the two variables **x** and **y** using the operator **>**. The returned value is **true** because **x** has a greater value than **y**. Similarly, in the next rows the results from the other 5 comparison operators, used to compare the variables **x** and **y**, are printed.

Assignment Operators

The operator for **assigning value to a variable** is "**=**" (the character for mathematical equation). The syntax used for assigning value is as it follows:

```
operand1 = literal, expression or operand2;
```

Assignment Operators – Example

Here is an example to show the usage of the assignment operator:

```
int x = 6;
string helloString = "Hello string.";
int y = x;
```

In the example we assign value 6 to the variable x. On the second line we assign a text literal to the variable **helloString**, and on the third line we copy the value of the variable **x** to the variable **y**.

Cascade Assignment

The assignment operator can be used in **cascade** (more than once in the same expression). In this case assignments are carried out consecutively from right to left. Here's an example:

```
int x, y, z;
x = y = z = 25;
```

On the first line in the example we initialize three variables and on the second line we assign them the value 25.

The assignment operator in C# is "=", while the comparison operator is "==". The exchange of the two operators is a common error when we are writing code. Be careful not to confuse the comparison operator and the assignment operator as they look very similar.

Compound Assignment Operators

Except the assignment operator there are also **compound assignment operators**. They help to reduce the volume of the code by typing two operations together with an operator: operation and assignment. Compound operators have the following syntax:

```
operand1 operator = operand2;
```

The upper expression is like the following:

```
operand1 = operand1 operator operand2;
```

Here is an example of a compound operator for assignment:

```
int x = 2;
int y = 4;

x *= y; // Same as x = x * y;
Console.WriteLine(x); // 8
```

The most commonly used compound assignment operators are **+=** (adds value of **operand2** to **operand1**), **-=** (subtracts the value of the right operand from the value of the left one).Other compound assignment operators are ***=**, **/=** and **%=**.

The following example gives a good idea of how the compound assignment operators work:

```
int x = 6;
```

```
int y = 4;

Console.WriteLine(y *= 2);  // 8
int z = y = 3;              // y=3 and z=3

Console.WriteLine(z);       // 3
Console.WriteLine(x |= 1);  // 7
Console.WriteLine(x += 3);  // 10
Console.WriteLine(x /= 2);  // 5
```

In the example, first we create the variables **x** and **y** and assign them values 6 and 4. On the next line we print on the console **y**, after we have assigned it a new value using the operator ***=** and the literal 2.The result of the operation is 8. Further in the example we apply the other compound assignment operators and print the result on the console.

Conditional Operator ?:

The **conditional operator ?:** uses the Boolean value of an expression to determine which of two other expressions must be calculated and returned as a result. The operator works on three operands and that is why it is called ternary operator. The character "**?**" is placed between the first and second operand, and "**:**" is placed between the second and third operand. The first operand (or expression) must be **Boolean**, and the next two operands must be **of the same type**, such as numbers or strings.

The operator **?:** has the following syntax:

```
operand1 ? operand2 : operand3
```

It works like this: if **operand1** is set to **true**, the operator returns as a result **operand2**. Otherwise (if **operand1** is set to **false**), the operator returns as a result **operand3**.

During the execution, the value of the first argument is calculated. If it has value **true**, then the second (middle) argument is calculated and it is returned as a result. However, if the calculated result of the first argument is **false**, then the third (last) argument is calculated and it is returned as a result.

Conditional Operator "?:" – Example

The following example shows the usage of the operator "**?:**":

```
int a = 6;
int b = 4;
Console.WriteLine(a > b ? "a>b" : "b<=a"); // a>b
```

```
int num = a == b ? 1 : -1; // num will have value -1
```

Other Operators

So far we have examined arithmetic, logical and bitwise operators, the operator for concatenating strings, also the conditional operator **?:**. Besides them in C # there are several other operators worth mentioning.

The "." Operator

The **access operator** "**.**" (dot) is used to access the member fields or methods of a class or object. Example of usage of point operator:

```
Console.WriteLine(DateTime.Now); // Prints the date + time
```

Square Brackets [] Operator

Square brackets **[]** are used to **access elements of an array by index**, they are the so-called **indexer**. Indexers are also used for accessing characters in a string. Example:

```
int[] arr = { 1, 2, 3 };
Console.WriteLine(arr[0]); // 1
string str = "Hello";
Console.WriteLine(str[1]); // e
```

Brackets () Operator

Brackets **()** are used to **override the priority of execution** of expressions and operators. We have already seen how the brackets work.

Type Conversion Operator

The operator for type conversion **(type)** is used to **convert** a variable from one type to another. We will examine it in details in the section "Type Conversion".

Operator "as"

The operator **as** also is used for **type conversion** but invalid conversion returns null, not an exception.

Operator "new"

The **new** operator is used to **create and initialize new objects**. We will examine it in details in the chapter "Creating and Using Objects".

Operator "is"

The **is** operator is used to check whether an object is compatible with a given type (**check object's type**).

Operator "??"

The operator **??** is similar to the conditional operator **?:**. The difference is that it is placed between two operands and returns the left operand only if its value is not null, otherwise it returns the right operand. Example:

```
int? a = 5;
Console.WriteLine(a ?? -1); // 5
string name = null;
Console.WriteLine(name ?? "(no name)"); // (no name)
```

Other Operators – Examples

Here is an example that shows the operators we just explained:

```
int a = 6;
int b = 3;

Console.WriteLine(a + b / 2);                        // 7
Console.WriteLine((a + b) / 2);                      // 4

string s = "Beer";
Console.WriteLine(s is string);                      // True

string notNullString = s;
string nullString = null;
Console.WriteLine(nullString ?? "Unspecified");   // Unspecified
Console.WriteLine(notNullString ?? "Specified"); // Beer
```

Type Conversion and Casting

Generally, operators work over arguments with the same data type. However, C# has a wide variety of data types from which we can choose the most appropriate for a particular purpose. To perform an operation on variables of two different data types we need to convert both to the same data type. **Type conversion** (**typecasting**) can be **explicit** and **implicit**.

All expressions in C# have a type. This type can derive from the expression structure and the types, variables and literals used in it. It is possible to write an expression which type is inappropriate for the current context. In some cases this will lead to a compilation error, but in other cases the context can get a type that is similar or related to the type of the expression. In this case the program performs a **hidden type conversion**.

Specific conversion from type **S** to type **T** allows the expression of type **S** to be treated as an expression of type **T** during the execution of the program. In some cases this will require a validation of the transformation. Here are some examples:

- Conversion of type **object** to type **string** will require verification at runtime to ensure that the value is really an instance of type **string**.
- Conversion from **string** to **object** does not require any verification. The type **string** is an inheritor of the type **object** and can be converted to its base class without a risk of an error or data loss. We shall examine inheritance in details in the chapter "Object-Oriented Programming Principles".
- Conversion of type **int** to **long** can be made without verification during the execution, because there is no risk of data loss since the set of values of type **int** is a subset of values of type **long**.
- Conversion from type **double** to **long** requires conversion of 64-bit floating-point value to 64-bit integer. Depending on the value, data loss is possible and therefore it is necessary to convert the types explicitly.

In C# not all types can be converted to all other types, but only to some of them. For convenience, we shall group some of the possible transformations in C# according to their type into three categories:

- implicit conversion;
- explicit conversion;
- conversion to or from **string**;

Implicit Type Conversion

Implicit (hidden) type conversion is possible only when there is no risk of data loss during the conversion, i.e. when converting from a lower range type to a larger range (e.g. from **int** to **long**). To make an implicit conversion it is not necessary to use any operator and therefore such transformation is called implicit. The **implicit conversion** is done automatically by the compiler when you assign a value with lower range to a variable with larger range or if the expression has several types with different ranges. In such case the conversion is executed into the type with the highest range.

Implicit Type Conversion – Examples

Here is an example of implicit type conversion:

```
int myInt = 5;
Console.WriteLine(myInt); // 5

long myLong = myInt;
Console.WriteLine(myLong); // 5
```

```
Console.WriteLine(myLong + myInt); // 10
```

In the example we create a variable **myInt** of type **int** and assign it the value 5. After that we create a variable **myLong** of type **long** and assign it the value contained in **myInt**. The value stored in **myLong** is automatically converted from type **int** to type **long**. Finally, we output the result from adding the two variables. Because the variables are from different types they are automatically converted to the type with the greater range, i.e. to type **long** and the result that is printed on the console is **long** again. Indeed, the given parameter to the method **Console.WriteLine()** is of type **long**, but inside the method it will be converted again, this time to type **string**, so it can be printed on the console. This transformation is performed by the method **Long.ToString()**.

Possible Implicit Conversions

Here are some possible implicit conversions of primitive data types in C#:

- **sbyte** → **short**, **int**, **long**, **float**, **double**, **decimal**;
- **byte** → **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**;
- **short** → **int**, **long**, **float**, **double**, **decimal**;
- **ushort** → **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**;
- **char** → **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal** (although **char** is a character type in some cases it may be regarded as a number and have a numeric type of behavior, it can even participate in numeric expressions);
- **uint** → **long**, **ulong**, **float**, **double**, **decimal**;
- **int** → **long**, **float**, **double**, **decimal**;
- **long** → **float**, **double**, **decimal**;
- **ulong** → **float**, **double**, **decimal**;
- **float** → **double**.

There is **no data loss when converting types of smaller range to types with a larger range**. The numerical value remains the same after conversion. There are a few exceptions. When you convert type **int** to type **float** (32-bit values), the difference is that **int** uses all bits for a whole number, whereas **float** has a part of bits used for representation of a fractional part. Hence, loss of precision is possible because of rounding when conversion from **int** to **float** is made. The same applies for the conversion of 64-bit **long** to 64-bit **double**.

Explicit Type Conversion

Explicit type conversion is used whenever there is a possibility of data loss. When converting floating point type to integer type there is always a loss of data coming from the elimination of the fractional part and an **explicit conversion** is obligatory (e.g. **double** to **long**). To make such a conversion it is necessary to use the operator for data conversion (**type**). There may also be data loss when converting a type with a wider range to type with a narrower one (**double** to **float** or **long** to **int**).

Explicit Type Conversion – Example

The following example illustrates the use of explicit type conversion and data loss that may occur in some cases:

```
double myDouble = 5.1d;
Console.WriteLine(myDouble); // 5.1

long myLong = (long)myDouble;
Console.WriteLine(myLong); // 5

myDouble = 5e9d; // 5 * 10^9
Console.WriteLine(myDouble); // 5000000000

int myInt = (int)myDouble;
Console.WriteLine(myInt); // -2147483648
Console.WriteLine(int.MinValue); // -2147483648
```

In the first line of the example we assign a value 5.1 to the variable **myDouble**. After we convert (explicitly) to type **long** using the operator **(long)** and print on the console the variable **myLong** we see that the variable has lost its fractional part, because **long** is an integer. Then we assign to the real double precision variable **myDouble** the value 5 billion. Finally, we convert **myDouble** to **int** by the operator **(int)** and print variable **myInt**. The result is the same like when we print **int.MinValue** because **myDouble** contains a value bigger than the range of **int**.

It is not always possible to predict what the value of a variable will be after its scope overflows! Therefore, use sufficiently large types and be careful when switching to a "smaller" type.

Data Loss during Type Conversion

We will give an example for data loss during type conversion:

```
long myLong = long.MaxValue;
```

```
int myInt = (int)myLong;

Console.WriteLine(myLong); // 9223372036854775807
Console.WriteLine(myInt); // -1
```

The type conversion operator may also be used in case of an intentional implicit conversion. This contributes to the readability of code, reducing the chance for errors and it is considered good practice by many programmers.

Here are some more examples for type conversions:

```
float heightInMeters = 1.74f; // Explicit conversion
double maxHeight = heightInMeters; // Implicit
double minHeight = (double)heightInMeters; // Explicit
float actualHeight = (float)maxHeight; // Explicit

float maxHeightFloat = maxHeight; // Compilation error!
```

In the example above at the last line we have an expression that will generate a compilation error. This is because we try implicitly to convert type **double** to **float**, which can cause data loss. C# is a strongly typed programming language and does not allow such appropriation of values.

Forcing Overflow Exceptions during Casting

Sometimes it is convenient, instead of getting the wrong result, when a type overflows during switching from larger to smaller type, to get notification of the problem. This is done by the keyword **checked** which includes a **check for overflow in integer types**:

```
double d = 5e9d; // 5 * 10^9
Console.WriteLine(d); // 5000000000
int i = checked((int)d); // System.OverflowException
Console.WriteLine(i);
```

During the execution of the code fragment above an exception (i.e. notification of an error) of type **OverflowException** is raised. More information about the exceptions and the methods to catch and handle them can be found in the chapter "Exception Handling".

Possible Explicit Conversions

The explicit conversions between numeral types in C# are possible between any couple among the following types:

```
sbyte, byte, short, ushort, char, int, uint, long, ulong, float, double, decimal
```

In these conversions data can be lost, like data about the number size or information about its precision.

Notice that conversion to or from **string** is not possible through typecasting.

Conversion to String

If it is necessary we can convert any type of data, including the value **null**, to string. The conversion of strings is done automatically whenever you use the concatenation operator (**+**) and one of the arguments is not of type string. In this case the argument is **converted to a string** and the operator returns a new string representing the concatenation of the two strings.

Another way to convert different objects to type string is to call the method **ToString()** of the variable or the value. It is valid for all data types in .NET Framework. Even calling **3.ToString()** is fully valid in C# and the result will return the string **"3"**.

Conversion to String – Example

Let's take a look on several examples for converting different data types to string:

```
int a = 5;
int b = 7;

string sum = "Sum = " + (a + b);
Console.WriteLine(sum);

String incorrect = "Sum = " + a + b;
Console.WriteLine(incorrect);

Console.WriteLine(
  "Perimeter = " + 2 * (a + b) + ". Area = " + (a * b) + ".");
```

The result from the example is as follows:

```
Sum = 12
Sum = 57
Perimeter = 24. Area = 35.
```

From the results it is obvious, that concatenating a number to a character string returns in result the string followed by the text representation of the number. Note that the "**+**" for concatenating strings can cause **unpleasant effects** on the addition of numbers, because it has equal priority with the operator "**+**" for mathematical addition. Unless the priorities of the operations are changed by placing the brackets, they will always be executed from left to right.

More details about converting from and to **string** we will look at the chapter "Console Input and Output".

Expressions

Much of the program's work is the calculation of expressions. **Expressions are sequences of operators, literals and variables** that are calculated to a value of some type (number, string, object or other type). Here are some examples of expressions:

```
int r = (150-20) / 2 + 5;

// Expression for calculating the surface of the circle
double surface = Math.PI * r * r;

// Expression for calculating the perimeter of the circle
double perimeter = 2 * Math.PI * r;

Console.WriteLine(r);
Console.WriteLine(surface);
Console.WriteLine(perimeter);
```

In the example three expressions are defined. The first expression calculates the radius of a circle. The second calculates the area of a circle, and the last one finds the perimeter. Here is the result from the fragment above:

```
70
15393.80400259
439.822971502571
```

Side Effects of Expressions

The calculation of the expression can have **side effects**, because the expression can contain embedded assignment operators, can cause increasing or decreasing of the value and calling methods. Here is an example of such a side effect:

```
int a = 5;
int b = ++a;

Console.WriteLine(a); // 6
Console.WriteLine(b); // 6
```

Expressions, Data Types and Operator Priorities

When writing expressions, the data types and the behavior of the used operators should be considered. Ignoring this can lead to unexpected results. Here are some simple examples:

```
// First example
double d = 1 / 2;
Console.WriteLine(d); // 0, not 0.5

// Second example
double half = (double)1 / 2;
Console.WriteLine(half); // 0.5
```

In the first example, an expression divides two integers (written this way, 1 and two are integers) and assigns the result to a variable of type **double**. The result may be unexpected for some people, but that is because they are ignoring the fact that in this case the operator "**/**" works over integers and the result is an integer obtained by cutting the fractional part.

The second example shows that if we want to do division with fractions in the result, it is necessary to convert to **float** or **double** at least one of the operands. In this scenario the division is no longer integer and the result is correct.

Division by Zero

Another interesting example is **division by 0**. Most programmers think that division by 0 is an invalid operation and causes an error at runtime (exception) but this is actually true only for integer division by 0. Here is an example, which shows that fractional division by **0** is **Infinity** or **NaN**:

```
int num = 1;
double denum = 0; // The value is 0.0 (real number)
int zeroInt = (int) denum; // The value is 0 (integer number)
Console.WriteLine(num / denum); // Infinity
Console.WriteLine(denum / denum); // NaN
Console.WriteLine(zeroInt / zeroInt); // DivideByZeroException
```

Using Brackets to Make the Code Clear

When working with expressions it is important to **use brackets** whenever there is the slightest doubt about the priorities of the operations. Here is an example that shows how useful the brackets are:

```
double incorrect = (double)((1 + 2) / 4);
Console.WriteLine(incorrect); // 0
```

```
double correct = ((double)(1 + 2)) / 4;
Console.WriteLine(correct); // 0.75

Console.WriteLine("2 + 3 = " + 2 + 3); // 2 + 3 = 23
Console.WriteLine("2 + 3 = " + (2 + 3)); // 2 + 3 = 5
```

Exercises

1. Write an expression that checks whether an integer is **odd or even**.
2. Write a Boolean expression that checks whether a given integer is **divisible by both 5 and 7**, without a remainder.
3. Write an expression that looks for a given integer if its **third digit** (right to left) is 7.
4. Write an expression that checks whether the **third bit** in a given integer is 1 or 0.
5. Write an expression that calculates the **area of a trapezoid** by given sides **a**, **b** and height **h**.
6. Write a program that prints on the console the **perimeter and the area of a rectangle** by given side and height entered by the user.
7. The gravitational field of the Moon is approximately 17% of that on the Earth. Write a program that calculates the **weight of a man on the moon** by a given weight on the Earth.
8. Write an expression that checks for a given point {x, y} if it is **within the circle** K({0, 0}, R=5). Explanation: the point {0, 0} is the center of the circle and 5 is the radius.
9. Write an expression that checks for given point {x, y} if it is **within the circle** K({0, 0}, R=5) and **out of the rectangle** [{-1, 1}, {5, 5}]. Clarification: for the rectangle the lower left and the upper right corners are given.
10. Write a program that takes as input a **four-digit number** in format **abcd** (e.g. 2011) and performs the following actions:
    - Calculates the sum of the digits (in our example 2+0+1+1 = 4).
    - Prints on the console the number in reversed order: **dcba** (in our example 1102).
    - Puts the last digit in the first position: **dabc** (in our example 1201).
    - Exchanges the second and the third digits: **acbd** (in our example 2101).

11. We are given a number **n** and a position **p**. Write a sequence of operations that prints the value of **the bit on the position p** in the number (0 or 1). Example: **n**=35, **p**=5 -> 1. Another example: n=35, **p**=6 -> 0.

12. Write a Boolean expression that checks if the bit on position **p** in the integer **v** has the value 1. Example v=5, **p**=1 -> `false`.

13. We are given the number **n**, the value **v** (**v** = 0 or 1) and the position **p**. write a sequence of operations that changes the value of **n**, so the bit on the position **p** has the value of **v**. Example: n=35, p=5, v=0 -> n=3. Another example: n=35, p=2, v=1 -> n=39.

14. Write a program that checks if a given number **n** (1 < **n** < 100) is a **prime number** (i.e. it is divisible without remainder only to itself and 1).

15. * Write a program that **exchanges the values of the bits** on positions 3, 4 and 5 with bits on positions 24, 25 and 26 of a given 32-bit unsigned integer.

16. * Write a program that **exchanges bits** {p, p+1, …, p+k-1} with bits {q, q+1, …, q+k-1} of a given 32-bit unsigned integer.

Solutions and Guidelines

1. Take the **remainder of dividing the number by 2** and check if it is **`0`** or **1** (respectively the number is odd or even). **Use % operator** to calculate the remainder of integer division.

2. Use a logical "AND" (**`&&`** operator) and the remainder operation **`%`** in division. You can also solve the problem by only one test: the division of 35 (think why).

3. Divide the number by 100 and save it in a new variable, which then divide by 10 and take the remainder. The remainder of the division by 10 is the third digit of the original number. Check if it is equal to 7.

4. Use **bitwise "AND"** on the current number and the number that has 1 only in the third bit (i.e. number 8, if bits start counting from 0). If the returned result is different from 0 the third bit is 1:

```
int num = 25;
bool bit3 = (num & 8) != 0;
```

5. The formula for **trapezoid surface** is: **`S = (a + b) * h / 2`**.

6. Search the Internet for **how to read integers** from the console and use the formula for rectangle area calculation. If you have difficulties see instructions on the next problem.

7. Use the following code to **read the number from the console**:

```
Console.Write("Enter number: ");
int number = Convert.ToInt32(Console.ReadLine());
```

Then **multiply by 0.17** and print it.

8. Use the **Pythagorean Theorem** $a^2 + b^2 = c^2$. The point is inside the circle when **(x*x) + (y*y) ≤ 5*5**.

9. Use the code from the previous task and **add a check for the rectangle**. A point is inside a rectangle with walls parallel to the axes, when in the same time it is right of the left wall, left of the right wall, down from the top wall and above the bottom wall.

10. To get the individual **digits of the number** you can divide by 10 and take the remainder of the division by 10:

```
int a = num % 10;
int b = (num / 10) % 10;
int c = (num / 100) % 10;
int d = (num / 1000) % 10;
```

11. Use **bitwise operations**:

```
int n = 35; // 00100011
int p = 6;
int i = 1; // 00000001
int mask = i << p; // Move the 1-st bit left by p positions

// If i & mask are positive then the p-th bit of n is 1
Console.WriteLine((n & mask) != 0 ? 1 : 0);
```

12. The task is similar to the previous one.

13. Use bitwise operations by analogy with the previous two problems. You can reset the bit at position **p** in the number **n** as follows:

```
n = n & (~(1 << p));
```

You can set bits in the **unit** at position **p** in the number **n** as follows:

```
n = n | (1 << p);
```

Think how you can combine the above two hints.

14. Read about **loops** in the Internet or in the chapter "Loops". Use a loop and check the number for divisibility by all integers from 1 to the square root of the number. Since **n < 100**, you can find in advance all prime numbers from 1 to 100 and checks the input over them. The **prime**

numbers in the range [1…100] are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 and 97.

15. Use 3 times a combination of **getting and setting a bit at a given position**. The first exchange is given below:

```
int bit3 = (num >> 3) & 1;
int bit24 = (num >> 24) & 1;
num = num & (~(1 << 24)) | (bit3 << 24);
num = num & (~(1 << 3)) | (bit24 << 3);
```

16. Extend the solution of the previous problem to perform a **sequence of bit exchanges in a loop**. Read about loops in the chapter "Loops".

Chapter 4. Console Input and Output

In This Chapter

In this chapter we will get familiar with the **console** as a tool for **data input and output**. We will explain what it is, when and how to use it, and how most programming languages access the console. We will get familiar with some of the features in C# for user interaction: reading text and numbers from the console and printing text and numbers. We will also examine the main streams for input-output operations **`Console.In`**, **`Console.Out`** and **`Console.Error`**, the **`Console`** and the usage of **format strings** for printing data in various formats.

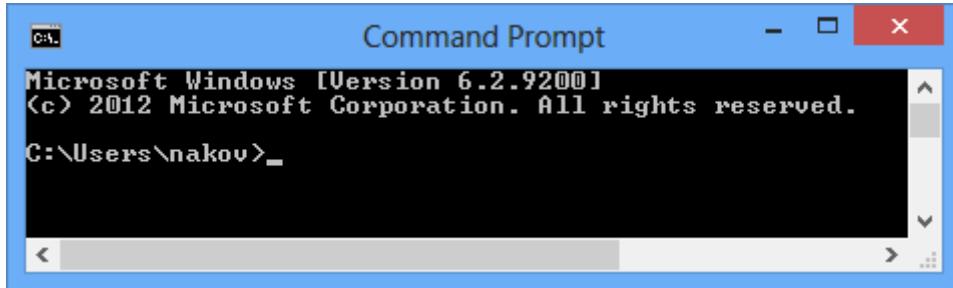
What Is the Console?

The **Console** is a window of the operating system through which users can interact with system programs of the operating system or with other console applications. The interaction consists of text input from the **standard input** (usually keyboard) or text display on the **standard output** (usually on the computer screen). These actions are also known as **input-output operations**. The text written on the console brings some information and is a sequence of characters sent by one or more programs.

For each console application the operating system connects input and output devices. By default these are the keyboard and the screen but they can be redirected to a file or other devices.

Communication between the User and the Program

A lot of programs communicate in some way with the user. This is necessary for the user in order to give instructions to them. Modern communication methods are many and various: they can be through **graphical** or **web-based interface**, **console** or others. As we mentioned one of the tools for communication between programs and users is the console, which is becoming less and less used. This is because the modern user interface concepts are more convenient and intuitive to work with, from a user's perspective.

When to Use the Console?

In some cases the console remains an irreplaceable tool for communication with the user. One of these cases is when writing **small and simple programs** where it is necessary to focus the attention on the specific problem to be solved, rather than the elegant representation of the result to the user. Then a simple solution is used for entering or printing a result, such as input-output console. Another use case is when we want to test a small piece of code for a larger application. Due to simplicity of the operation of the console application we can isolate this part of the code easily and comfortably without having to go through a complex user interface and a number of screens to get to the desired code for testing.

How to Launch the Console?

Each operating system has its own way to launch the console. On Windows for example, it can be done in the following way:

Start -> (All) Programs -> Accessories -> Command Prompt

After starting the console a black screen (this color can be changed) like the following should appear:

When starting the console the home directory of the current user (in this case the username is **nakov**) is used as a current directory and this is displayed as a guide for the user.

Console can be launched through pressing the Start button and typing "cmd" in the search box and pressing [Enter] (on Windows Vista, Windows 7 and later). For Windows XP, go through the sequence Start -> Run… ->, type in "cmd" and press [Enter].

For simplified visualization of the results from now on in this chapter instead of a console screenshot we will use the form:

```
Results from console
```

More about Consoles

The system console is **the black window** shown above **which displays text information**. It can display text strings and has a cursor, which moves to the right after each character is printed. After the cursor passes through the last column of the console (usually it has 80 columns), it moves to the beginning of the next line. If the cursor passes through the last line, the console scrolls its content upwards and shows a new empty line below the last line.

Programs in Windows can be console-based, desktop-based, Web-based and other. The **console-based programs** use the console for their input and output. The desktop-based programs use graphical user interface (GUI). The Web-based programs have Web-based user interface. In this book we will write console-based programs almost all the time, so their input will be read from the keyboard and their output will be printed in the console.

Some console-based programs expect the users to enter text, numbers and other data, and this is usually done through the keyboard.

The console in Windows is often associated with the system **command interpreter**, also called the "**Command Prompt**" or "**shell**" or which is a console-based program in the operating system, which provides access to system commands as well as a wide range of programs, which are part of the operating system or are additionally installed to it.

The word "**shell**" means "wrap" and has a meaning of a wrapper between the user and the inside of the operating system.

The so called operating system "shells" can be split into two main categories according to the type of interface they can provide to the operating system:

- CLI – Command Line Interface – is a console for commands (such as **cmd.exe** in Windows and **bash** in Linux).

- GUI – Graphical User Interface – is a graphical work environment (such as Windows Explorer).

For both types the main purpose of the shell is **to run other programs** with which the user works although most of the interpreters also support some advanced features such as the opportunity to examine the content of directories with files.

Each operating system has its own command interpreter that has its own commands.

For example, when starting Windows console, we run the so-called Windows **command interpreter** in it (**cmd.exe**) that executes system programs and commands in interactive mode. For example, the command **dir** shows the files in the current directory:

```
Command Prompt

C:\>dir
 Volume in drive C is Windows 2003
 Volume Serial Number is CCAB-5301

 Directory of C:\

10.02.2013  04:47    <DIR>          inetpub
17.03.2013  23:47    <DIR>          IntroCSharp
24.02.2013  12:55    <DIR>          Program Files
13.03.2013  23:57    <DIR>          Program Files (x86)
17.03.2013  22:49    <DIR>          Trash
02.03.2013  01:21    <DIR>          Users
13.03.2013  08:00    <DIR>          Windows
               0 File(s)              0 bytes
               7 Dir(s)  19 574 104 064 bytes free

C:\>_
```

Basic Console Commands

We will take a look at some basic commands in the Windows standard **command prompt**, which is useful for finding and launching programs.

Windows Console Commands

The command interpreter running in the console is also called "**Command Prompt**" or "**MS-DOS Prompt**" (in older versions of Windows). We will take a look at some basic commands for this interpreter:

Command	Description
dir	Displays the content of the current directory.
cd <directory name>	Changes the current directory.
mkdir <directory name>	Creates a new directory in the current one.
rmdir <directory name>	Deletes an existing directory.
type <file name>	Prints file content.
copy <src file> <destination file>	Copies one file into another.

Here is an example of multiple commands executed in the Windows command shell. The result of the commands' execution is displayed on the console:

```
C:\Documents and Settings\User1>cd "D:\Project2009\C# Book"

C:\Documents and Settings\User1>D:
```

```
D:\Project2008\C# Book>dir
 Volume in drive D has no label.
 Volume Serial Number is B43A-B0D6

 Directory of D:\Project2009\C# Book

26.12.2009   12:24    <DIR>          .
26.12.2009   12:24    <DIR>          ..
26.12.2009   12:23           537 600 Chapter-4-Console-Input-
Output.doc
26.12.2009   12:23    <DIR>          Test Folder
26.12.2009   12:24                 0 Test.txt
             2 File(s)        537 600 bytes
             3 Dir(s)  24 154 062 848 bytes free

D:\Project2009\C# Book>
```

Standard Input-Output

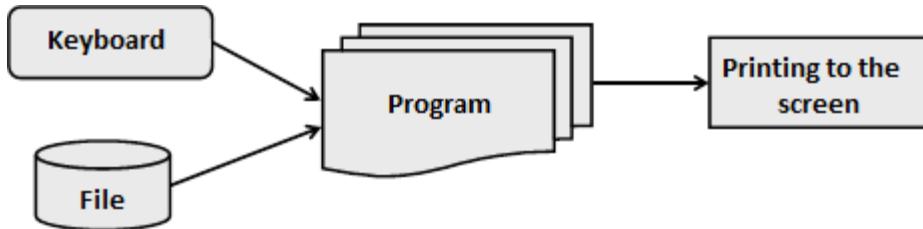
The standard input-output also known as "**Standard I/O**" is a system input-output mechanism created since the UNIX operating systems was developed many years ago. Special peripheral devices for input and output are used, through which data can be input and output.

When the program is in mode of accepting information and expects action by the user, there is a blinking cursor on the console showing that the system is waiting for command entering.

Later we will see how we can write C# programs that expect input data to be entered from the console.

Printing to the Console

In most programming languages printing and reading the information from the console is implemented in similar ways and the most of the solutions are based on the concept of "**standard input**" and "**standard output**".

Standard Input and Standard Output

The operating system is required to define **standard input-output mechanisms** for user interaction. When starting a given console program, system code running at the initialization of the program is responsible for opening (closing) of streams to the allocated by the operating system mechanisms for input-output. This system code initializes the program abstraction for user interaction embedded in the respective programming language. In this way, the application started can automatically read the user

input from the **standard input stream** (in C# this is **Console.In**), print information on the **standard output stream** (in C# this is **Console.Out**) and can signal for problem situations in the **standard error stream** (in C# this is **Console.Error**).

The concept of the streams will be later examined in details. For now we will focus on the theoretical basis related to the program input and output in C#.

Devices for Console Input and Output

Besides the keyboard an application **input** can come from many other places, such as file, microphone, barcode reader and others. The **output** of a program may be on the console (on the screen), as well as in a file or another output device, such as a printer:

We will show a basic example that illustrates text printing to the console through the abstraction for accessing the standard input and standard output provided to us by C#:

```
Console.Out.WriteLine("Hello World");
```

The result of the above code execution would be the following:

```
Hello World
```

Console.Out Stream

System.Console class has different properties and methods (classes are considered in details in the chapter "Creating and Using Objects") which are used to read and display text on the console as well as its formatting. Among them there are three properties that make impression and they are related to data entering and displaying, namely the **Console.Out**, **Console.In** and **Console.Error**. They provide access to the standard streams for printing on the console, for reading from the console and to the error messages reporting stream accordingly. Although we could use them directly, the other methods of **System.Console** give us the convenience for working with input-output console operations and actually most often these properties are ignored. However it is good to remember that this part of the console functionality is working on these streams. If needed, we can replace the default input / output / error streams at runtime by using the methods **Console.SetOut(…)**, **Console.SetIn(…)** and **Console.SetError(…)** respectively.

Now we will examine the most commonly used methods for text printing on the console.

Using Console.Write(…) and Console.WriteLine(…)

Work with these methods is easy because they can print all the basic types (string, numeric and primitive types).

Here are some examples of printing various types of data:

```
// Print String
Console.WriteLine("Hello World");

// Print int
Console.WriteLine(5);

// Print double
Console.WriteLine(3.14159265358979);
```

The result of this code execution looks like this:

```
Hello World
5
3.14159265358979
```

As we see by using **Console.WriteLine(…)** it is possible to print various data types because for each type there is a predefined version of the method**WriteLine(…)** in the **Console** class.

The difference between **Write(…)** and **WriteLine(…)** is that the **Write(…)** method prints on the console what it is provided between the parentheses but does nothing in addition while the method **WriteLine(…)** means directly "write line". This method does what the **Write(…)** one does but in addition goes to a new line. In fact the method does not print a new line but simply puts a **"command" for moving** cursor to the position where the new line starts (this command consists of the character **\r** followed by **\n**).

Here is an example, which illustrates the difference between **Write(…)** and **WriteLine(…)**:

```
Console.WriteLine("I love");
Console.Write("this ");
Console.Write("Book!");
```

The output of this example is:

```
I love
this Book!
```

We notice that the output of this example is printed on two lines, even though the code is on three. This happens because on the first line of code we use **WriteLine(…)** which prints "**I love**" and then goes to a new line. In the next two lines of the code uses the **Write(…)** method, which prints without going on a new line and thus the words "**this**" and "**Book!**" remain on the same line.

Concatenation of Strings

In general C# does not allow the use of operators over string objects. The only exception to this rule is the addition operation (**+**) which **concatenates (joins) two strings** and returns as result a new string. This allows chaining of concatenate (**+**) operations one after another in a sequence. The next example represents concatenation of three strings.

```
string age = "twenty six";
string text = "He is " + age + " years old.";
Console.WriteLine(text);
```

The result of this code execution is again a string:

```
He is twenty six years old.
```

Concatenation of Mixed Types

What happens when we want to print larger and more complex text, which consists of different types? Until now we used versions of the method **WriteLine(…)** for a specific type. Is it necessary when we want to print different types at once to use different versions of the method **WriteLine(…)** for each of these types? The answer to this question is "no" because in C# we can unite text and other data (for instance, numeric) by using the "**+**" operator. The following example is like the previous but in it the years (**age**) are from integer type:

```
int age = 26;
string text = "He is " + age + " years old.";
Console.WriteLine(text);
```

In the example is concatenation and printing on the screen performed. The result of the example is the following:

```
He is 26 years old.
```

On the second line of the example code we see that a concatenation of the string "**He is**" and the integer type "**age**" is performed. We are trying to **combine two different types**. This is possible because of the presence of the following important rule.

When a string is involved in concatenation with any other type the result is always a string.

From the rule it is clear that the result of **"He is " + age** is again a string and then the result is added to the last part of the expression **" years old."**. So after calling a chain of **+** operators ultimately the result is a string and thus the string version of the method **WriteLine(…)** is invoked.

For short the above example can be written as follows:

```
int age = 26;
Console.WriteLine("He is " + age + " years old.");
```

Some Features of String Concatenation

There are some interesting situations with concatenation (addition) of strings that you need to know and be careful about because they lead to errors. The following example represents a surprising behavior of the code:

```
string s = "Four: " + 2 + 2;
Console.WriteLine(s);
// Four: 22

string s1 = "Four: " + (2 + 2);
Console.WriteLine(s1);
// Four: 4
```

As seen from the example the operators' execution order (see chapter "Operator and Expressions") is of great importance! In our example first the concatenation of "**Four: **" to "**2**" is performed and **the result of the operation is string**. After that, another concatenation with the second number is performed and the obtained unexpected result is "**Four: 22**" instead of the expected "**Four: 4**". This is because the operations are performed from left to right and in this scenario a string participates in each of them.

In order to avoid this **unpleasant situation** we can use parentheses that will change the order of operators' execution can be used to achieve the desired result. Parentheses are operators with highest priority and make the execution of the operation "addition" of the two numbers happen before the concatenation with the string on the left. Thus first the addition of the two numbers is done and then they are concatenated with the string.

This mistake is very common for beginner programmers because they do not consider that string concatenation is performed from left to right because the addition of numbers is of the same priority than as concatenation.

When you concatenate strings and also sum numbers, use parentheses to specify the correct order of operations. Otherwise they are executed from left to right.

Formatted Output with Write(…) and WriteLine(…)

For printing long and elaborate series of elements, special options (also known as overloads) of the methods **Write(…)** and **WriteLine(…)** have been introduced. These options have a completely different concept than the standard methods for printing in C#. Their main idea is to adopt a special string, formatted with special formatting characters and list of values, which should be substituted in place of "the format specifiers". Here is how **Write(…)** is defined in the standard C# libraries:

```
public static void Write(string format, object arg0,
  object arg1, object arg2, object arg3, …);
```

Formatted Output – Examples

The following example prints twice the same thing but in different ways:

```
string str = "Hello World!";

// Print (the normal way)
Console.Write(str);

// Print (through formatting string)
Console.Write("{0}", str);
```

The result of this example execution is:

```
Hello World!Hello World!
```

We see as a result "**Hello, World!**" twice on one line. This is because there is no printing of a new line in the program.

First we print the string in a well-known way in order to see the difference with the other approach. The second printing is the formatting **Write(…)** and the first argument is the format string. In this case **{0}** means to put the first argument after the formatting string in the place of **{0}**. The expression **{0}** is called a **placeholder**, i.e. a place that will be replaced by a specific value while printing.

The next example will further explain this concept:

```
string name = "John";
int age = 18;
```

```
string town = "Seattle";
Console.Write(
  "{0} is {1} years old from {2}!\n", name, age, town);
```

The result of this example execution is as follows:

```
John is 18 years old from Seattle!
```

From the signature of this **Write(…)** version we saw that the first argument is the format string. Following is a series of arguments, which are placed where we have a number enclosed in curly brackets. The expression **{0}** means to put in its place the **first** of the arguments submitted after the format string (in this case **name**). Next is **{1}** which means to replace with the **second** of the arguments (**age**). The last placeholder is **{2}**, which means to replace with the next parameter (**town**). Last is **\n**, which is a special character that indicates moving to a new line.

It is appropriate to mention that actually the new line command on **Windows** is **\r\n**, and on **Unix-based operating systems** – **\n**. When working with the console it does not matter that we use only **\n** because the standard input stream considers **\n** as **\r\n** but if we write into a file, for example, using only **\n** is wrong (on Windows).

Composite Formatting

The methods for formatted output of the **Console** class use the so-called **composite formatting feature**. The composite formatting is used for console printing as well as in certain operations with strings. We examined the composite formatting in the simplest of its kind in the previous example but it has significantly bigger potential than what we have seen so far. Basically the composite formatting uses two things: **composite formatting string** and **series of arguments**, which are replaced in certain places in the string.

Composite Formatting String

The composite formatting string is a mixture of normal text and **formatting items**. In formatting the normal text remains the same as in the string and the places of formatting items are replaced by the values of the respective arguments printed according to certain rules. These rules are specified using the syntax of formatting items.

Formatting Items

The formatting items provide the possibility for powerful control over the displayed value and therefore can obtain very complicated form. The following formation scheme represents the general syntax of **formatting items**:

```
{index[,alignment][:formatString]}
```

As we notice the formatting item begins with an opening curly bracket **{** and ends with a closing curly bracket **}**. The content between the brackets is divided into three components of which only the **index** component is mandatory. Now we will examine each of them separately.

Index Component

The **index** component is an integer and indicates **the position** of the argument from the argument list. The first argument is indicated by "**0**", the second by "**1**", etc. The **composite formatting string** allows having multiple formatting items that relate to one and same argument. In this case **index** component of these items is one and the same number. There is no restriction on the sequence of arguments' calling. For example, we could use the following formatting string:

```
Console.Write(
  "{1} is {0} years old from {3}!", 18, "John", 0, "Seattle");
```

In cases where some of the arguments are not referenced by any of the formatting items, those arguments are simply **ignored** and do not play a role. However it is good to remove such arguments from the list of arguments because they introduce unnecessary complexity and may lead to confusion.

In the opposite case, when a formatting item refers an argument that does not exist in the list of arguments, an **exception is thrown**. This may occur, for example, if we have formatting placeholder **{4}** and we submitted a list of only two arguments.

Alignment Component

The **alignment** component is optional and indicates **the string alignment**. It is a **positive or negative integer** and the positive values indicate alignment to the right and the negative – alignment to the left. The value of the number indicates the number of positions in which to align the number. If the string we want to represent has a length greater than or equal to the value of the number, then this number is ignored. If it is less, however, the unfilled positions are filled in with spaces.

For example, let's try the following formatting:

```
Console.WriteLine("{0,6}", 123);
Console.WriteLine("{0,6}", 1234);
Console.WriteLine("{0,6}", 12);
Console.Write("{0,-6}", 123);
Console.WriteLine("--end");
```

It will output the following result:

```
   123
```

```
  1234
    12
123   --end
```

If we decide to use the alignment component, we must separate it from the **`index`** component by a comma as it is done in the example above.

The "formatString" Component

This component specifies the specific formatting of the string. It varies depending on the type of argument. There are three main types of **`formatString`** components:

- for numerical types of arguments
- for arguments of type date (**`DateTime`**)
- for arguments of type enumeration (listed types)

Format String Components for Numbers

This type **`formatString`** component has two subtypes: standard-defined formats and user-defined formats (custom format strings).

Standard Formats for Numbers

These formats are defined by one of several **format specifiers**, which are letters with particular importance. After the format specifier there can be a positive integer called **precision**, which has a different meaning for the different specifiers. When it affects the number of decimal places after the decimal point, the result is rounded. The following table describes specifiers and their **precision** meaning:

Specifier	Description
"C" or "c"	Indicates the **currency** and the result will be displayed along with the currency sign for the current “culture” (for example, English). The precision indicates the number of decimal places after the decimal point.
"D" or "d"	An **integer number**. The precision indicates the minimum number of characters for representing the string and, if necessary, zeroes are supplemented in the beginning.
"E" or "e"	**Exponential notation**. The precision indicates the number of places after the decimal point.
"F" or "f"	**Integer or decimal number**. The precision indicates the number of signs after the decimal point.

"N" or "n"	Equivalent to "F" but represents also the corresponding separator for thousands, millions, etc. (for example, in the English language often the number "**1000**" is represented as "**1,000**" – with comma between the number 1 and the zeroes).
"P" or "p"	Percentage: it will multiply the number by 100 and will display the **percent** character upfront. The precision indicates the number of signs after the decimal point.
"X" or "x"	Displays the number in **hexadecimal** numeral system. It works only for integer numbers. The precision indicates minimum numbers of signs to display the string as the missing ones are supplemented with zeroes at the beginning.

Part of the formatting is determined by **the current "culture" settings**, which are taken by default from the regional settings of the operating system. "The cultures" are set of rules that are valid for a given language or a given country and that indicate which character is to be used as decimal separator, how the currency is displayed, etc. For example, for the Japanese "culture" the currency is displayed by adding "**¥**" after the amount, while for the American "culture", the character "**$**" is displayed before the amount. For Bulgarian currency is suffixed by " **лв.**".

Standard Formats for Numbers – Example

Let's see a few examples of usage of the specifiers represented in the table above. In the code below we assume the regional settings are **Bulgarian** so the currency will be printed in Bulgarian, the decimal separator will be "**,**" and the thousands separator will be space (the regional settings can be changed from Control Panel in Windows):

StandardNumericFormats.cs

```
class StandardNumericFormats
{
  static void Main()
  {
    Console.WriteLine("{0:C2}", 123.456);
    //Output: 123,46 лв.
    Console.WriteLine("{0:D6}", -1234);
    //Output: -001234
    Console.WriteLine("{0:E2}", 123);
    //Output: 1,23E+002
    Console.WriteLine("{0:F2}", -123.456);
    //Output: -123,46
```

```
        Console.WriteLine("{0:N2}", 1234567.8);
        //Output: 1 234 567,80
        Console.WriteLine("{0:P}", 0.456);
        //Output: 45,60 %
        Console.WriteLine("{0:X}", 254);
        //Output: FE
    }
}
```

If we run the same code with English (United States) culture, the output will be as follows:

```
$123.46
-001234
1.23E+002
-123.46
1,234,567.80
45.60 %
FE
```

Custom Formats for Numbers

All formats that are not standard are assigned to the user (custom) formats. For the **custom formats** are again defined a set of specifiers and the difference with the standard formats is that a number of specifiers can be used (in standard formats only a single specifier is used). The following table lists various specifiers and their meaning:

Specifier	Description
0	Indicates a digit. If at this position of the result a digit is missing, a zero is written instead.
#	Indicates a digit. Does not print anything if at this position in the result a digit is missing.
.	Decimal separator for the respective "culture".
,	Thousands separator for the respective "culture".
%	Multiplies the result by 100 and prints the character for percent.
E0 or **E+0** or **E-0**	Indicates an exponential notation. The number of zeroes indicates the number of signs of the exponent. The sign "**+**" means that we always want to represent also the number's sign while minus means to display the sign only if the value is negative.

There are many characteristics regarding the use of custom formats for numbers, but they will not be discussed here. You may find more information in MSDN. Here are some simple examples that illustrate how to use custom formatting strings (the output is given for the U.S. culture):

CustomNumericFormats.cs

```
class CustomNumericFormats
{
  static void Main()
  {
    Console.WriteLine("{0:0.00}", 1);
    //Output: 1.00
    Console.WriteLine("{0:#.##}", 0.234);
    //Output: .23
    Console.WriteLine("{0:#####}", 12345.67);
    //Output: 12346
    Console.WriteLine("{0:(0#) ### ## ##}", 29342525);
    //Output: (02) 934 25 25
    Console.WriteLine("{0:%##}", 0.234);
    //Output: %23
  }
}
```

Format String Components for Dates

When formatting dates we again have separation of standard and custom formats.

Standard Defined Date Formats

Since the standard defined formats are many we will list only few of them. The rest can be easily checked on MSDN.

Specifier	Format (for English (United States) "culture")
d	2/27/2012
D	February 27, 2012
t	17:30 (hour)
T	17:30:22 (hour)
Y or y	February 2012 (only month and year)

Custom Date Formats

Similar to custom formats for numbers here we have multiple format specifiers and we can combine several of them. Since here are many

specifiers we will show only some of them, which we will use to demonstrate how to use **custom formats for dates**. Consider the following table:

Specifiers	Format (for English (United States) "culture")
d	Day – from 1 to 31
dd	Day – from 01 to 31
M	Month – from 1 to 12
MM	Month – from 01 to 12
yy	The last two digits of the year (from 00 to 99)
yyyy	Year written in 4 digits (e.g. 2012)
hh	Hour – from 00 to 11
HH	Hour – from 00 to 23
m	Minutes – from 0 to 59
mm	Minutes – from 00 to 59
s	Seconds – from 0 to 59
ss	Seconds – from 00 to 59

When using these specifiers we can insert different separators between the different parts of the date, such as "**.**" or "**/**". Here are few examples:

```
DateTime d = new DateTime(2012, 02, 27, 17, 30, 22);
Console.WriteLine("{0:dd/MM/yyyy HH:mm:ss}", d);
Console.WriteLine("{0:d.MM.yy}", d);
```

Execution of these examples gives the following result for the U.K. culture:

```
27/02/2012 17:30:22
27.02.12
```

Note that the result can vary depending on the current culture. For example if we run the same code in the Bulgarian culture, the result will be different:

```
27.02.2012 17:30:22
27.02.12
```

Format String Enumeration Components

Enumerations (listed types) are data types that can take as value one of several predefined possible values (e.g. the seven days of the week). We will examine them in details in the chapter "Defining Classes".

In enumerations there is very little to be formatted. Four standard format specifiers are defined:

Specifier	Format
`G or g`	Represents enumeration as a string.
`D or d`	Represents enumeration as a number.
`X or x`	Represents enumeration as a number in hexadecimal numeral system and with eight digits.

Here are some examples:

```
Console.WriteLine("{0:G}", DayOfWeek.Wednesday);
Console.WriteLine("{0:D}", DayOfWeek.Wednesday);
Console.WriteLine("{0:X}", DayOfWeek.Wednesday);
```

While executing the above code we get the following result:

```
Wednesday
3
00000003
```

Formatting Strings and Localization

When using format strings it is possible one and same program to print different values depending on the **localization settings** for the operating system. For example, when printing the month from a given date if the current localization is English it will print in English, for example “August”, while if the localization is French it will print in French, for example "Août".

When launching a console application it automatically retrieves the operating system localization (culture settings) and uses it for reading and writing formatted data (like numbers, dates, currency, etc.).

Localization in .NET is also called "culture" and can be changed manually by the class **`System.Globalization.CultureInfo`**. Here is an example in which we print a number and a date by the **U.S.** and **Bulgarian** localization:

CultureInfoExample.cs

```
using System;
using System.Threading;
using System.Globalization;

class CultureInfoExample
{
  static void Main()
```

```
  {
    DateTime d = new DateTime(2012, 02, 27, 17, 30, 22);

    Thread.CurrentThread.CurrentCulture =
      CultureInfo.GetCultureInfo("en-US");
    Console.WriteLine("{0:N}", 1234.56);
    Console.WriteLine("{0:D}", d);

    Thread.CurrentThread.CurrentCulture =
      CultureInfo.GetCultureInfo("bg-BG");
    Console.WriteLine("{0:N}", 1234.56);
    Console.WriteLine("{0:D}", d);
  }
}
```

When starting the example the following result is obtained:

```
1,234.56
Monday, February 27, 2012
1 234,56
27 Февруари 2012 г.
```

Console Input

As in the beginning of this chapter we explained, the most suitable for small applications is the console communication because it is easiest to implement. **The standard input device** is the part of the operating system that controls from where the program will receive its input data. By default "the standard input device" reads its input from a driver "attached" to the keyboard. This can be changed and the standard input can be redirected to another location, for example to a file, but this is rarely done.

Each programming language has a mechanism for reading and writing to the console. The object that controls the standard input stream in **C#**, is **Console.In**.

From the console we can read different data:

- text;
- other types after parsing the text;

Actually for reading the standard input stream **Console.In** is rarely used directly. The class **Console** provides two methods **Console.Read()** and **Console.ReadLine()** that run on this stream and usually reading from the console is done by them.

Reading through Console.ReadLine()

The method **Console.ReadLine()** provides great convenience for reading from console. How does it work? When this method is invoked, the program prevents its work and wait for input from the console. The user enters some string on the console and presses the **[Enter]** key. At this moment the console understands that the user has finished entering and reads the string. The method **Console.ReadLine()** returns as result the string entered by the user. Now perhaps it is clear why this method has this name.

The following example demonstrates the operation of **Console.ReadLine()**:

UsingReadLine.cs

```
class UsingReadLine
{
  static void Main()
  {
    Console.Write("Please enter your first name: ");
    string firstName = Console.ReadLine();

    Console.Write("Please enter your last name: ");
    string lastName = Console.ReadLine();

    Console.WriteLine("Hello, {0} {1}!", firstName, lastName);
  }
}

// Output:  Please enter your first name: John
//          Please enter your last name: Smith
//          Hello, John Smith!
```

We see how easy it is to read text from the console by using the method **Console.ReadLine()**:

- We print some text in the console, which asks for a user name (this is only for the convenience of the user and is not obligatory).
- We execute reading of an entire line from the console using the method **ReadLine()**. This leads to blocking the program until the user enters some text and presses [Enter].
- Then we repeat these two steps for the last name.
- Once we have gathered the necessary information we print it on the console.

Reading through Console.Read()

The method **Read()** behaves slightly different than **ReadLine()**. As a beginning it reads only one character and not the entire line. The other significant difference is that the method does not return directly the read character but its code. If we want to use the result as a character we must convert it to a character or use the method **Convert.ToChar()** on it. There is one important characteristic: **the character is read only when the [Enter] key is pressed**. Then the entire string written on the console is transferred to the buffer of the standard input string and the method **Read()** reads the first character of it. In subsequent invocations of the method if the buffer is not empty (i.e. there are already entered in but still unread characters) then the program execution will not stop and wait, but will directly read the next character from the buffer and thus until the buffer is empty. Only then the program will wait again for a user input if **Read()** is called again. Here is an example:

UsingRead.cs

```
class UsingRead
{
   static void Main()
   {
      int codeRead = 0;
      do
      {
         codeRead = Console.Read();
         if (codeRead != 0)
         {
            Console.Write((char)codeRead);
         }
      }
      while (codeRead != 10);
   }
}
```

This program reads one line entered by the user and prints it character by character. This is possible due to a small trick – we are previously aware that the [Enter] key actually enters two characters in the buffer. These are the "**carriage return**" code (**Unicode 13**) followed by the "**linefeed**" code (**Unicode 10**). In order to understand that one line is finished we are looking for a character with code **10** in the Unicode table. Thus the program reads only one line and exits the loop.

We should mention that the method **Console.Read()** is rarely used in practice if there is an alternative to use **Console.ReadLine()**. The reason for this is that the possibility of mistaking with **Console.Read()** is much greater

than if we choose an alternative approach and the code will most likely be unnecessarily complicated.

Reading Numbers

Reading numbers from the console in C# **is not done directly**. In order to read a number we should have previously read the input as a string (using **ReadLine()**) and then convert this string to a number. The operation of converting a string into another type is called **parsing**. All primitive types have methods for parsing. We will give a simple example for reading and parsing of numbers:

ReadingNumbers.cs

```
class ReadingNumbers
{
  static void Main()
  {
    Console.Write("a = ");
    int a = int.Parse(Console.ReadLine());

    Console.Write("b = ");
    int b = int.Parse(Console.ReadLine());

    Console.WriteLine("{0} + {1} = {2}", a, b, a + b);
    Console.WriteLine("{0} * {1} = {2}", a, b, a * b);

    Console.Write("f = ");
    double f = double.Parse(Console.ReadLine());
    Console.WriteLine("{0} * {1} / {2} = {3}",
      a, b, f, a * b / f);
  }
}
```

The result of program execution might be as follows (provided that we enter 5, 6 and 7.5 as input):

```
a = 5
b = 6
5 + 6 = 11
5 * 6 = 30
f = 7.5
5 * 6 / 7.5 = 4
```

In this particular example the specific thing is that we use **parsing methods of numerical types** and when wrong a result is passed (such as text) this

will cause an error (exception) **System.FormatException**. This is especially true when reading real numbers, because the delimiter used between the whole and fractional part is different in various cultures and depends on regional settings of the operating system.

The separator for floating point numbers depends on the current language settings of the operating system (Regional and Language Options in Windows). In some systems as separator the character comma can be used, in others – point (dot). Entering a point (dot) instead of a comma will cause System.FormatException when the current language settings use comma.

The exceptions as a mechanism for reporting errors will be discussed in the chapter "Exception Handling". For now you can consider that when the program provides an error this is associated with the occurrence of an exception that prints detailed information about the error on the console. For example, let's suppose that the regional settings of the computer are Bulgarian and we execute the following code:

```
Console.Write("Enter a floating-point number: ");
string line = Console.ReadLine();
double number = double.Parse(line);
Console.WriteLine("You entered: {0}", number);
```

If we enter the number "3.14" (with a wrong decimal separator for the Bulgarian settings) we will get the following **exception** (error message):

```
Unhandled Exception: System.FormatException: Input string was
not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles
options, NumberBuffer& number, NumberFormatInfo info, Boolean
parseDecimal)
   at System.Number.ParseDouble(String value, NumberStyles
options, NumberFormatInfo numfmt)
   at System.Double.Parse(String s, NumberStyles style,
NumberFormatInfo info)
   at System.Double.Parse(String s)
   at ConsoleApplication.Program.Main() in
C:\Projects\IntroCSharpBook\ConsoleExample\Program.cs:line 14
```

Parsing Numbers Conditionally

When parsing a string to a number using the method **Int32.Parse(string)** or by **Convert.ToInt32(string)** if the submitted string is not a number we

get an exception. Sometimes it is necessary to catch the failed parsing and to print an error message or to ask the user to enter in a new value.

Interception of an incorrectly entered number when parsing a sting can be done in two ways:

- by **catching exceptions** (see the chapter "Exception Handling");
- by **conditional parsing** (using the method **TryParse(…)**).

Let's consider the **conditional parsing of numbers** in .NET Framework. The method **Int32.TryParse(…)** accepts two parameters – a parsing string and a variable to record the result of parsing. If the parsing is successful the method returns value **true**. For greater clarity, let's consider an example:

```
string str = Console.ReadLine();
int intValue;
bool parseSuccess = Int32.TryParse(str, out intValue);
Console.WriteLine(parseSuccess ?
  "The square of the number is " + intValue * intValue + "."
  : "Invalid number!");
```

In the example, conditional parsing of a string entered from the console to the integer type **Int32** is performed. If we enter as input "**2**", parsing will be successful so the result of **TryParse()** will be **true**, and the parsed number will be recorded in the variable **intValue** and on the console the squared number will be printed:

```
Result: The square of the number is 4.
```

If we try to parse an invalid number such as "**abc**", **TryParse()** will return **false** as a result and the user that will be notified that he has entered an invalid number:

```
Invalid number!
```

Note that the method **TryParse()** as a result of its work **returns simultaneously two values**: the parsed number (as an output parameter) and a Boolean value as a result of the method invocation. Returning multiple values at once is possible because one of the values is returned as an **output parameter** (**out** parameter). The output parameters return value in a predefined for the purpose variable coinciding with their type. When calling a method the output parameters must be preceded by the keyword **out**.

Reading by Console.ReadKey()

The method **Console.ReadKey()** waits for key pressing on the console and reads its character equivalent without the need of pressing **[Enter]**. The result of invoking **ReadKey()** is **information about the pressed key** (or

more accurately a **key combination**) as an object of type **ConsoleKeyInfo**. The obtained object contains the character that is entered by the pressed key combination (property **KeyChar**) along with information about the keys [Shift], [Ctrl] and [Alt] (property **Modifiers**). For example, if we press [Shift+A] we will read a capital letter '**A**' while in the Modifiers property we will have the **Shift** flag. Here is an example:

```
ConsoleKeyInfo key = Console.ReadKey();
Console.WriteLine();
Console.WriteLine("Character entered: " + key.KeyChar);
Console.WriteLine("Special keys: " + key.Modifiers);
```

If we execute the program and press **[Shift+A]**, we will obtain the following result:

```
A
Character entered: A
Special keys: Shift
```

Simplified Reading of Numbers through Nakov.IO.Cin

There is no standard easy way to read several numbers, located on the same line, separated by a space. In C# and .NET Framework we need to read a string, split it into tokens using the space as separator and parse the obtained tokens to extract the numbers. In other languages and platforms like C++ we can directly read numbers, characters and text from the console without parsing. This is not available in C# but we can use an external library or class.

The **standard library Nakov.IO.Cin** provides a simplified way to read numbers from the console. You can read about it from the blog of its author Svetlin Nakov: http://www.nakov.com/blog/2011/11/23/cin-class-for-csharp-read-from-console-nakov-io-cin/. Once we have copied the file **Cin.cs** from **Nakov.IO.Cin** into our Visual Studio C# project, we could write code like this:

```
using Nakov.IO;
…
int x = Cin.NextInt();
double y = Cin.NextDouble();
decimal d = Cin.NextDecimal();
Console.WriteLine("Result: {0} {1} {2}", x, y, d);
```

If we execute the code, we can enter 3 numbers by putting any amount of whitespace separators between them. For example we can enter the first number, two spaces, the second number, a new line + space and the last number + space. The **numbers will be read correctly** and the output will be as follows:

```
3   2.5
 3.58
Result: 3 2.5 3.58
```

Console Input and Output – Examples

We will consider few more examples of console input and output that will show us some interesting techniques.

Printing a Letter

Next is a practical example representing console input and formatted text in the form of a letter:

PrintingLetter.cs

```
class PrintingLetter
{
  static void Main()
  {
    Console.Write("Enter person name: ");
    string person = Console.ReadLine();

    Console.Write("Enter book name: ");
    string book = Console.ReadLine();

    string from = "Authors Team";

    Console.WriteLine("  Dear {0},", person);
    Console.Write("We are pleased to inform " +
      "you that \"{1}\" is the best Bulgarian book. {2}" +
      "The authors of the book wish you good luck {0}!{2}",
      person, book, Environment.NewLine);

    Console.WriteLine("  Yours,");
    Console.WriteLine("  {0}", from);
  }
}
```

The result of the execution of the above program could be the following:

```
Enter person name: Readers
Enter book name: Introduction to programming with C#
  Dear Readers,
We are pleased to inform you that "Introduction to programming
```

```
with C#" is the best Bulgarian book.
The authors of the book wish you good luck Readers!
  Yours,
  Authors Team
```

In this example we have a letter template. The program "asks" a few questions to the user and reads from the console information needed to print the letter by replacing the formatting specifiers with the data filled in by the user.

Area of a Rectangle or a Triangle

We will consider another example: calculating of an area of a rectangle or a triangle.

CalculatingArea.cs

```
class CalculatingArea
{
  static void Main()
  {
    Console.WriteLine("This program calculates " +
    "the area of a rectangle or a triangle");

    Console.WriteLine("Enter a and b (for rectangle) " +
      "or a and h (for triangle): ");

    int a = int.Parse(Console.ReadLine());
    int b = int.Parse(Console.ReadLine());

    Console.WriteLine("Enter 1 for a rectangle or " +
        "2 for a triangle: ");

    int choice = int.Parse(Console.ReadLine());
    double area = (double) (a * b) / choice;
    Console.WriteLine("The area of your figure is " + area);
  }
}
```

The result of the above example's execution is as follows:

```
This program calculates the area of a rectangle or a triangle
Enter a and b (for rectangle) or a and h (for triangle):
5
4
```

```
Enter 1 for a rectangle or 2 for a triangle:
2
The area of your figure is 10
```

Exercises

1. Write a program that **reads** from the console **three numbers** of type **int** and prints their sum.
2. Write a program that **reads** from the console the **radius** "**r**" of a circle and prints its **perimeter and area**.
3. A given company has name, address, phone number, fax number, web site and manager. The manager has name, surname and phone number. Write a program that **reads information about the company** and its manager and then **prints it** on the console.
4. Write a program that **prints three numbers in three virtual columns** on the console. Each column should have a width of 10 characters and the numbers should be **left aligned**. The first number should be an integer in **hexadecimal**; the second should be **fractional positive**; and the third – a **negative fraction**. The last two numbers have to be rounded to the second decimal place.
5. Write a program that reads from the console two integer numbers (**int**) and prints how many numbers between them exist, such that **the remainder of their division by 5 is 0**. Example: in the range (14, 25) there are 3 such numbers: 15, 20 and 25.
6. Write a program that reads two numbers from the console and **prints the greater of them**. Solve the problem without using conditional statements.
7. Write a program that **reads five integer numbers and prints their sum.** If an invalid number is entered the program should prompt the user to enter another number.
8. Write a program that reads five numbers from the console and prints the **greatest** of them.
9. Write a program that reads an integer number **n** from the console. After that reads **n** numbers from the console and prints their **sum**.
10. Write a program that reads an integer number **n** from the console and **prints all numbers in the range [1…n]**, each on a separate line.
11. Write a program that prints on the console the first 100 numbers in the **Fibonacci sequence**: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …
12. Write a program that calculates the **sum** (with **precision of 0.001**) of the following sequence: 1 + 1/2 - 1/3 + 1/4 - 1/5 + …

Solutions and Guidelines

1. Use the methods **Console.ReadLine()** and **Int32.Parse()**.

2. Use **Math.PI** constant and the well-known **geometric formulas**.

3. Format the text with **Write(…)** or **WriteLine(…)** similar to the example with the letter that we looked at.

4. Use the format strings explained in the "Composite Formatting" section and the method **Console.WriteLine()**. Below is a piece of the code:

```
int hexNum = 2013;
Console.WriteLine("|0x{0,-8:X}|", hexNum);
double fractNum = -1.856;
Console.WriteLine("|{0,-10:f2}|", fractNum);
```

5. There are two approaches for solving the problem:

   First approach: Use mathematical tricks for optimized calculation based on the fact that **every fifth number is divisible by 5**. Think how to implement this correctly and about the borderline cases.

   The **second approach** is easier but it works slower. With a **for-loop** each number within the given range can be checked. You should read in Internet or in the chapter "Loops" how to use **for**-loops.

6. Since the problem requires a solution, which **does not use conditional statements**, you should use a different approach. Two possible solutions of the problem include the use of functions of class **Math**. The **greater** of the two numbers you can find with the function **Math.Max(a, b)** and the **smaller** with **Math.Min(a, b)**.

   Another solution to the problem includes usage of the function for taking the absolute value of a number **Math.Abs(a)**:

```
int a = 2011;
int b = 1990;
Console.WriteLine("Greater: {0}", (a + b + Math.Abs(a-b)) / 2);
Console.WriteLine("Smaller: {0}", (a + b - Math.Abs(a-b)) / 2);
```

   The third solution uses **bitwise operations**:

```
int a = 1990;
int b = 2011;
int max = a - ((a - b) & ((a - b) >> 31));
Console.WriteLine(max);
```

   There is another solution which is partially correct because it uses a **hidden conditional statement** (the ternary **?:** operator):

```
int a = 1990;
int b = 2013;
int max = a > b ? a : b;
Console.WriteLine(max);
```

7. You can read the numbers in **five different variables** and finally sum them and print the obtained sum. Note that the sum of 5 **int** values may not fit in the **int** type so you should use **long**.

   Another approach is **using loops**. When parsing the consecutive numbers use **conditional parsing** with **TryParse(…)**. When an invalid number is entered, repeat reading of the number. You can do this through **while** loop with an appropriate exit condition. To avoid repetitive code you can explore the **for**-loops from the chapter "Loops".

8. You can use the **comparison statement** "**if**" (you can read about it on the Internet or from the chapter "Conditional Statements"). To avoid repeating code you can use the looping construct "**for**" (you could read about it online or in the chapter "Loops").

9. You should use a **for-loop** (see the chapter "Loops"). Read the numbers one after another and accumulate their sum in a variable, which then display on the console at the end.

10. Use a combination of **loops** (see the chapter "Loops") and the methods **Console.ReadLine()**, **Console.WriteLine()** and **Int32.Parse()**.

11. More about the **Fibonacci sequence** can be found in Wikipedia at: http://en.wikipedia.org/wiki/Fibonacci_sequence. For the solution of the problem use 2 temporary variables in which store the last 2 calculated values and with a loop calculate the rest (each subsequent number in the sequence is a sum of the last two). Use a **for-loop** to implement the repeating logic (see the chapter "Loops").

12. Accumulate the **sum of the sequence** in a variable inside a **while-loop** (see the chapter "Loops"). At each step compare the old sum with the new sum. If the difference between the two sums **Math.Abs(current_sum – old_sum)** is less than the required precision (**0.001**), the calculation should finish because the difference is constantly decreasing and the precision is constantly increasing at each step of the loop. The expected result is **1.307**.

Chapter 5. Conditional Statements

In This Chapter

In this chapter we will cover the **conditional statements in C#**, which we can use to execute different actions depending on a given condition. We will explain the syntax of the conditional operators **`if`** and **`if-else`** with suitable examples and explain the practical application of the operator for selection **`switch-case`**.

We will focus on the **best practices** to be followed in order to achieve a better programming style when using nested or other types of conditional statements.

Comparison Operators and Boolean Expressions

In the following section we will recall the **basic comparison operators** in the C# language. They are important, because we use them to describe conditions in our conditional statements.

Comparison Operators

There are several comparisons operators in C#, which are used to compare pairs of integers, floating-point numbers, characters, strings and other types:

Operator	Action
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to

Comparison operators can be used to compare expressions such as two numbers, two numerical expressions, or a number and a variable. The result of the comparison is a Boolean value (**`true`** or **`false`**).

Let’s look at an example of using comparisons:

```
int weight = 700;
Console.WriteLine(weight >= 500); // True

char gender = 'm';
Console.WriteLine(gender <= 'f'); // False

double colorWaveLength = 1.630;
Console.WriteLine(colorWaveLength > 1.621); // True

int a = 5;
int b = 7;
bool condition = (b > a) && (a + b < a * b);
Console.WriteLine(condition); // True

Console.WriteLine('B' == 'A' + 1); // True
```

In the sample code we perform a comparison between numbers and between characters. The numbers are compared by size while characters are compared by their lexicographical order (the operation uses the Unicode numbers for the corresponding characters).

As seen in the example, the type **char behaves like a number** and can be subtracted, added and compared to numbers freely. However, this should be used cautiously as it could make the code difficult to read and understand.

By running the example we will produce the following output:

```
True
False
True
True
True
```

In C# several types of data that can be compared:

- numbers (**int**, **long**, **float**, **double**, **ushort**, **decimal**, …)
- characters (**char**)
- Booleans (**bool**)
- References to objects, also known as object pointers (**string**, **object**, arrays and others)

Every comparison can affect two numbers, two **bool** values, or two object references. It is allowed to **compare expressions of different types**, like an integer with a floating-point number for example. However, not every pair of data types can be compared directly. For example, we cannot compare a string with a number.

Comparison of Integers and Characters

When comparing integers and characters, we directly compare their binary representation in memory i.e. we **compare their values**. For example, if we compare two numbers of type **int**, we will compare the values of their respective series of 4 bytes. Here is one example for integer and character comparisons:

```
Console.WriteLine("char 'a' == 'a'? " + ('a' == 'a'));  // True
Console.WriteLine("char 'a' == 'b'? " + ('a' == 'b'));  // False
Console.WriteLine("5 != 6? " + (5 != 6));               // True
Console.WriteLine("5.0 == 5L? " + (5.0 == 5L));         // True
Console.WriteLine("true == false? " + (true == false)); // False
```

The result of the example is as follows:

```
char 'a' == 'a'? True
char 'a' == 'b'? False
5 != 6? True
5.0 == 5L? True
true == false? False
```

Comparison of References to Objects

In .NET Framework there are reference data types that do not contain their value (unlike the value types), but contain the address of the memory in the heap where their value is located. Strings, arrays and classes are such types. They behave like a pointer to some value and can have the value **null**, i.e. no value. When comparing reference type variables, we **compare the addresses** they hold, i.e. we check whether they point to the same location in the memory, i.e. to the same object.

Two object pointers (references) can refer to the same object or to different objects, or one of them can point to nowhere (to have **null** value). In the following example we create two variables that point to the same value (object) in the heap.

```
string str = "beer";
string anotherStr = str;
```

After executing the source code above, the two variables **str** and **anotherStr** will point to the same object (**string** with value "beer"), which is located at some address in the heap (managed heap).

We can check whether the **variables point to the same object** with the comparison operator (**==**). For most reference types this operator does not compare the content of the objects but rather checks if they point at the same

location in memory, i.e. if they are one and the same object. The size comparisons (**<**, **>**, **<=** and **>=**) are not applicable for object type variables.

The following example illustrates the comparison of references to objects:

```
string str = "beer";
string anotherStr = str;
string thirdStr = "be" + 'e' + 'r';
Console.WriteLine("str = {0}", str);
Console.WriteLine("anotherStr = {0}", anotherStr);
Console.WriteLine("thirdStr = {0}", thirdStr);
Console.WriteLine(str == anotherStr); // True - same object
Console.WriteLine(str == thirdStr); // True - equal objects
Console.WriteLine((object)str == (object)anotherStr); // True
Console.WriteLine((object)str == (object)thirdStr); // False
```

If we execute the sample code, we will get the following result:

```
str = beer
anotherStr = beer
thirdStr = beer
True
True
True
False
```

Because the strings used in the example (instances of the class **System.String**, defined by the keyword **string** in C#) are of reference type, their values are set as objects in the heap. The two objects **str** and **thirdStr** have equal values, but are different objects, located at separate addresses in the memory. The variable **anotherStr** is also reference type and gets the address (the reference) of **str**, i.e. points to the existing object **str**. So by the comparison of the variables **str** and **anotherStr**, it appears that they are one and the same object and are equal. The result of the comparison between **str** and **thirdStr** is also equality, because the operator **==** compares the strings by value and not by address (a very useful exception to the rule for comparison by address). However, if we convert the three variables to objects and then compare them, we will get a comparison of the addresses in the heap where their values are located and the result will be different.

This above example shows that the operator **== has a special behavior when comparing strings**, but for the rest of the reference types (like arrays or classes) it applies comparison by address.

You will learn more about the class **String** and the comparison of strings in the chapter about "Strings".

Logical Operators

Let's recall the logical operators in C#. They are often used to construct logical (Boolean) expressions. The logical operators are: **&&**, **||**, **!** and **^**.

Logical Operators && and ||

The logical operators **&&** (logical AND) and **||** (logical OR) are only used on Boolean expressions (values of type **bool**). In order for the result – of comparing two expressions with the operator **&&** – to be **true** (true), both operands must have the value **true**. For instance:

```
bool result = (2 < 3) && (3 < 4);
```

This expression is "true", because both the operands: (2 < 3) and (3 < 4) are "true". The logical operator **&&** is also called **short-circuit**, because it does not lose time in additional unnecessary calculations. It evaluates the left part of the expression (the first operand) and if the result is **false**, it does not lose time for evaluating the second operand – it's not possible the end result to be "true" when the first operand is not "true". For this reason it is also called **short-circuit logical operator "and"**.

Similarly, the operator || returns true if at least one of the two operands has the value "true". Example:

```
bool result = (2 < 3) || (1 == 2);
```

This example is "true", because its first operand is "true". Just like the **&&** operator, the calculation is done fast – if the first operand is **true**, the second is not calculated at all, as the result is already known. It is also called **short-circuit logical operator "or"**.

Logical Operators & and |

The operators for comparison **&** and **|** are similar to **&&** and **||**, respectively. The difference lies in the fact that both operands are calculated one after the other, although the final result is known in advance. That's why these comparison operators are also known as **full-circuit logical operators** and are used very rarely.

For instance, when two operands are compared with **&** and the first one is evaluated "false", the calculation of the second operand is still executed. The result is clearly "false". Likewise, when two operands are compared with **|** and the first one is "true", we still evaluate the second operand and the final result is nevertheless "true".

We must not confuse the Boolean operators **&** and **|** with the bitwise operators **&** and **|**. Although they are written in the same way, they take different arguments (Boolean or integer expressions) and return different result (**bool** or integer) and their actions are not identical.

Logical Operators ^ and !

The ^ operator, also known as **exclusive OR (XOR)**, belongs to the full-circuit operators, because both operands are calculated one after the other. The result of applying the operator is **true if exactly one of the operands is true, but not both simultaneously**. Otherwise the result is **false**. Here is an example:

```
Console.WriteLine("Exclusive OR: "+ ((2 < 3) ^ (4 > 3)));
```

The result is as follows:

```
Exclusive OR: False
```

The previous expression is evaluated as false, because both operands: (2 <3) and (4 > 3) are true.

The operator **!** returns the reversed value of the Boolean expression to which it is attached. Example:

```
bool value = !(7 == 5); // True
Console.WriteLine(value);
```

The above expression can be read as "the opposite of the truth of the phrase "**7 == 5**". The result of this pattern is **True** (the opposite of **False**). Note that when we print the value **true** it is displayed on the console as "**True**" (with capital letter). This "defect" comes from the VB.NET language that also runs in .NET Framework.

Conditional Statements "if" and "if-else"

After reviewing how to compare expressions, we will continue with conditional statements, which will allow us to implement programming logic.

Conditional statements if and **if-else** are conditional control statements. Because of them the program can behave differently based on a defined condition checked during the execution of the statement.

Conditional Statement "if"

The main format of the conditional statements **if** is as follows:

```
if (Boolean expression)
{
  Body of the conditional statement;
}
```

It includes: **if**-clause, Boolean expression and body of the conditional statement.

The **Boolean expression** can be a Boolean variable or Boolean logical expression. Boolean expressions cannot be integer (unlike other programming languages like C and C++).

The **body of the statement** is the part locked between the curly brackets: **{}**. It may consist of one or more operations (statements). When there are several operations, we have a complex block operator, i.e. series of commands that follow one after the other, enclosed in curly brackets.

The expression in the brackets which follows the keyword **if** must return the Boolean value **true** or **false**. If the expression is calculated to the value **true**, then the body of a conditional statement is executed. If the result is **false**, then the operators in the body will be skipped.

Conditional Statement "if" – Example

Let's take a look at an example of using a conditional statement **if**:

```
static void Main()
{
  Console.WriteLine("Enter two numbers.");
  Console.Write("Enter first number: ");
  int firstNumber = int.Parse(Console.ReadLine());
  Console.Write("Enter second number: ");
  int secondNumber = int.Parse(Console.ReadLine());
  int biggerNumber = firstNumber;
  if (secondNumber > firstNumber)
  {
    biggerNumber = secondNumber;
  }
  Console.WriteLine("The bigger number is: {0}", biggerNumber);
}
```

If we start the example and enter the numbers 4 and 5 we will get the following result:

```
Enter two numbers.
Enter first number: 4
Enter second number: 5
The bigger number is: 5
```

Conditional Statement "if" and Curly Brackets

If we have only one operator in the body of the **if**-statement, the curly brackets denoting the body of the conditional operator may be omitted, as shown below. However, it is a good practice to use them even if we have only one operator. This will make the code is more readable.

Here is an example of omitting the curly brackets which leading to confusion:

```
int a = 6;
if (a > 5)
  Console.WriteLine("The variable is greater than 5.");
  Console.WriteLine("This code will always execute!");
// Bad practice: misleading code
```

In this example the code is misleadingly formatted and creates the impression that both printing statements are part of the body of the **if**-block. In fact, this is true only for the first one.

	Always put curly brackets { } for the body of "if" blocks even if they consist of only one operator!

Conditional Statement "if-else"

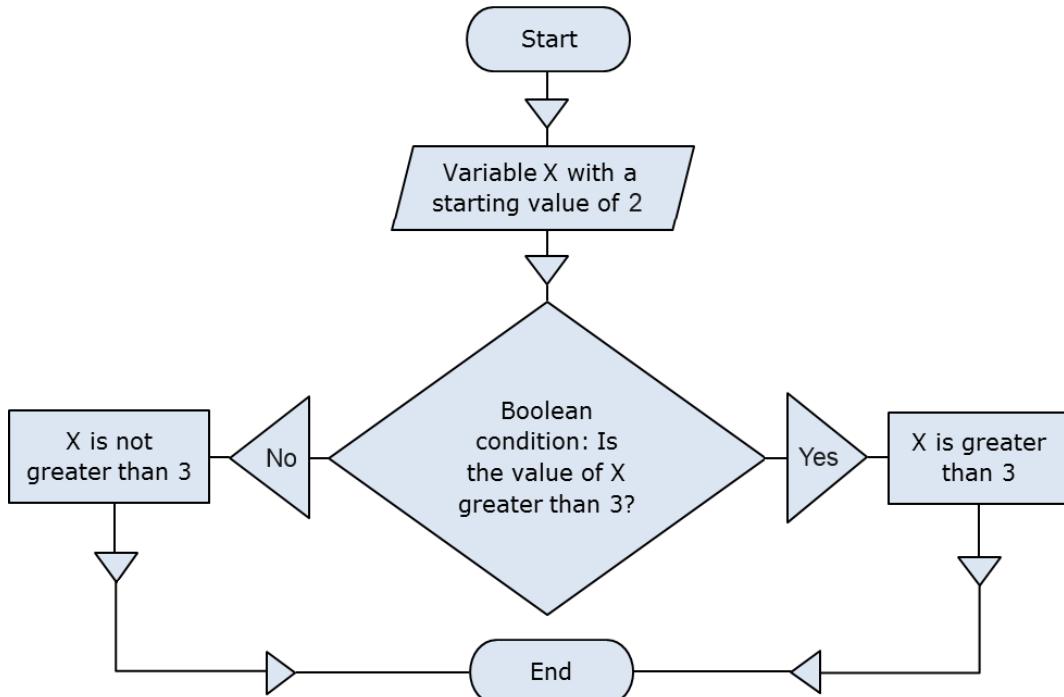
In C#, as in most of the programming languages there is a conditional statement with **else** clause: the **if-else** statement. Its format is the following:

```
if (Boolean expression)
{
  Body of the conditional statement;
}
else
{
  Body of the else statement;
}
```

The format of the **if-else** structure consists of the reserved word **if**, Boolean expression, body of a conditional statement, reserved word **else** and **else**-body statement. The body of **else**-structure may consist of one or more operators, enclosed in curly brackets, same as the body of a conditional statement.

This statement works as follows: the **expression in the brackets** (a Boolean expression) is calculated. The calculation result must be Boolean – **true** or **false**. Depending on the result there are two possible outcomes. If the Boolean expression is calculated to **true**, the **body of the conditional statement** is executed and the **else**-statement is omitted and its operators do not execute. Otherwise, if the Boolean expression is calculated to **false**, **the else-body is executed**, the main body of the conditional statement is omitted and the operators in it are not executed.

Conditional Statement "if-else" – Example

Let's take a look at the next example and illustrate how the **if-else** statement works:

```
static void Main()
{
   int x = 2;
   if (x > 3)
   {
      Console.WriteLine("x is greater than 3");
   }
   else
   {
      Console.WriteLine("x is not greater than 3");
   }
}
```

The program code can be interpreted as follows: if x>3, the result at the end is: "**x is greater than 3**", otherwise (**else**) the result is: "**x is not greater than 3**". In this case, since x=2, after the calculation of the Boolean expression the operator of the **else** structure will be executed. The result of the example is:

```
x is not greater than 3
```

The following scheme illustrates the process flow of this example:

Nested "if" Statements

Sometimes the programming logic in a program or an application needs to be represented by multiple **if**-structures contained in each other. We call them **nested if** or **nested if-else structures**.

We call nesting the placement of an **if** or **if-else** structure in the body of another **if** or **else** structure. In such situations every **else** clause corresponds to the closest previous **if** clause. This is how we understand which **else** clause relates to which **if** clause.

It's not a good practice to exceed three nested levels, i.e. we should not nest more than three conditional statements into one another. If for some reason we need to nest more than three structures, we should export a part of the code in a separate method (see chapter Methods).

Nested "if" Statements – Example

Here is an example of using nested **if** structures:

```
int first = 5;
int second = 3;

if (first == second)
{
  Console.WriteLine("These two numbers are equal.");
}
else
{
  if (first > second)
  {
    Console.WriteLine("The first number is greater.");
  }
  else
  {
    Console.WriteLine("The second number is greater.");
  }
}
```

In the example above we have two numbers and compare them in two steps: first we compare whether they are equal and if not, we compare again, to determine which one is the greater. Here is the result of the execution of the code above:

```
The first number is greater.
```

Sequences of "if-else-if-else-…"

Sometimes we need to use a **sequence of if structures**, where the **else** clause is a new **if** structure. If we use nested **if** structures, the code would be pushed too far to the right. That's why in such situations it is allowed to use a new **if** right after the **else**. It's even considered a good practice. Here is an example:

```
char ch = 'X';
if (ch == 'A' || ch == 'a')
{
  Console.WriteLine("Vowel [ei]");
}
else if (ch == 'E' || ch == 'e')
{
  Console.WriteLine("Vowel [i:]");
}
else if (ch == 'I' || ch == 'i')
{
  Console.WriteLine("Vowel [ai]");
}
else if (ch == 'O' || ch == 'o')
{
  Console.WriteLine("Vowel [ou]");
}
else if (ch == 'U' || ch == 'u')
{
  Console.WriteLine("Vowel [ju:]");
}
else
{
  Console.WriteLine("Consonant");
}
```

The program in the example makes a series of comparisons of a variable to check if it is one of the **vowels from the English alphabet**. Every following comparison is done only in case that the previous comparison was not true. In the end, if none of the **if**-conditions is not fulfilled, the last **else** clause is executed. Thus, the result of the example is as follows:

```
Consonant
```

Conditional "if" Statements – Good Practices

Here are some guidelines, which we recommend for writing if, structures:

- Use blocks, surrounded by curly brackets **{}** after **if** and **else** in order to avoid ambiguity
- Always format the code correctly by offsetting it with one tab inwards after **if** and **else**, for readability and avoiding ambiguity.
- Prefer **switch-case** structure to of a series of **if-else-if-else-…** structures or nested **if-else** statement, if possible. The construct **switch-case** we will cover in the next section.

Conditional Statement "switch-case"

In the following section we will cover the conditional statement **switch**. It is used for choosing among a list of possibilities.

How Does the "switch-case" Statement Work?

The structure **switch-case** chooses which part of the programming code to execute based on the calculated value of a certain expression (most often of integer type). The format of the structure for choosing an option is as follows:

```
switch (integer_selector)
{
   case integer_value_1:
      statements;
      break;
   case integer_value_2:
      statements;
      break;
   // …
   default:
      statements;
      break;
}
```

The **selector** is an expression returning a resulting value that can be compared, like a number or **string**. The **switch** operator compares the result of the selector to every value listed in the **case labels** in the body of the switch structure. If a match is found in a **case** label, the corresponding structure is executed (simple or complex). If no match is found, the **default** statement is executed (when such exists). The value of the selector must be calculated before comparing it to the values inside the **switch** structure. The labels should not have repeating values, they must be unique.

As it can be seen from the definition above, every **case** ends with the operator **break**, which ends the body of the **switch** structure. The C# compiler requires the word **break** at the end of each **case**-section containing code. If no code is found after a **case**-statement, the **break** can be omitted

and the execution passes to the next **case**-statement and continues until it finds a **break** operator. After the **default** structure **break** is obligatory.

It is not necessary for the **default** clause to be last, but it's recommended to put it at the end, and not in the middle of the **switch** structure.

Rules for Expressions in Switch

The **switch** statement is a clear way to implement selection among many options (namely, a choice among a few alternative ways for executing the code). It requires a selector, which is calculated to a certain value. The selector type could be an integer number, **char**, **string** or **enum**. If we want to use for example an array or a float as a selector, it will not work. For non-integer data types, we should use a series of **if** statements.

Using Multiple Labels

Using multiple labels is appropriate, when we want to execute the same structure in more than one case. Let's look at the following example:

```
int number = 6;
switch (number)
{
  case 1:
  case 4:
  case 6:
  case 8:
  case 10:
    Console.WriteLine("The number is not prime!"); break;
  case 2:
  case 3:
  case 5:
  case 7:
    Console.WriteLine("The number is prime!"); break;
  default:
    Console.WriteLine("Unknown number!"); break;
}
```

In the above example, we implement multiple labels by using **case** statements without **break** after them. In this case, first the integer value of the selector is calculated – that is **6**, and then this value is compared to every integer value in the **case** statements. When a match is found, the code block after it is executed. If no match is found, the **default** block is executed. The result of the example above is as follows:

```
The number is not prime!
```

Good Practices When Using "switch-case"

- A good practice when using the **switch** statement is to put the **default statement at the end**, in order to have easier to read code.
- It's good to place first the **cases**, which handle t**he most common situations**. **Case** statements, which handle situations occurring rarely, can be placed at the end of the structure.
- If the values in the **case** labels are integer, it's recommended that they be arranged in **ascending order**.
- If the values in the **case** labels are of character type, it's recommended that the **case** labels are **sorted alphabetically**.
- It's advisable to always use a **default** block to handle situations that cannot be processed in the normal operation of the program. If in the normal operation of the program the **default** block should not be reachable, you could put in it a **code reporting an error**.

Exercises

1. Write an **if**-statement that takes two integer variables and **exchanges** their values if the first one is greater than the second one.
2. Write a program that shows the sign (**+** or **-**) of the product of three real numbers, without calculating it. Use a sequence of **if** operators.
3. Write a program that finds the **biggest of three integers**, using nested **if** statements.
4. **Sort 3 real numbers** in descending order. Use nested **if** statements.
5. Write a program that asks for a digit (0-9), and depending on the input, **shows the digit as a word** (in English). Use a **switch** statement.
6. Write a program that gets the coefficients ***a***, ***b*** and ***c*** of a quadratic equation: $ax^2 + bx + c$, calculates and prints its real roots (if they exist). Quadratic equations may have 0, 1 or 2 real roots.
7. Write a program that finds the **greatest of given 5 numbers**.
8. Write a program that, depending on the user's choice, inputs **int**, **double** or **string** variable. If the variable is **int** or **double**, the program increases it by 1. If the variable is a **string**, the program appends "*" at the end. Print the result at the console. Use **switch** statement.
9. We are given 5 integer numbers. Write a program that finds those **subsets whose sum is 0**. Examples:
   - If we are given the numbers {3, -2, 1, 1, 8}, the sum of -2, 1 and 1 is 0.
   - If we are given the numbers {3, 1, -7, 35, 22}, there are no subsets with sum 0.

10. Write a program that applies **bonus points** to given scores in the range [1…9] by the following rules:
    - If the score is between 1 and 3, the program multiplies it by 10.
    - If the score is between 4 and 6, the program multiplies it by 100.
    - If the score is between 7 and 9, the program multiplies it by 1000.
    - If the score is 0 or more than 9, the program prints an error message.

11. * Write a program that **converts a number in the range [0…999] to words**, corresponding to the English pronunciation. Examples:
    - 0 --> "Zero"
    - 12 --> "Twelve"
    - 98 --> "Ninety eight"
    - 273 --> "Two hundred seventy three"
    - 400 --> "Four hundred"
    - 501 --> "Five hundred and one"
    - 711 --> "Seven hundred and eleven"

Solutions and Guidelines

1. Look at the section about if-statements.

2. A multiple of non-zero numbers has a positive product, if the **negative multiples are even number**. If the count of the negative numbers is odd, the product is negative. If at least one of the numbers is zero, the product is also zero. Use a counter **`negativeNumbersCount`** to keep the **number of negative numbers**. Check each number whether it is negative and change the counter accordingly. If some of the numbers is 0, print "**`0`**" as result (the zero has no sign). Otherwise print "**`+`**" or "**`-`**" depending on the condition (**`negativeNumbersCount % 2 == 0`**).

3. Use **nested `if`-statements**, first checking the first two numbers then checking the bigger of them with the third.

4. First **find the smallest** of the three numbers, and then **swap it with the first one**. Then check if the second is greater than the third number and if yes, swap them too.

   Another approach is to check all possible orders of the numbers with a series of **`if-else`** checks: **`a≤b≤c`**, **`a≤c≤b`**, **`b≤a≤c`**, **`b≤c≤a`**, **`c≤a≤b`** and **`c≤b≤a`**.

   A **more complicated** and more general solution of this problem is to put the numbers in an array and use the **`Array.Sort(…)`** method. You may read about arrays in the chapter "Arrays".

5. Just use a **`switch`** statement to check for all possible digits.

6. From math it is known, that a **quadratic equation** may have one or two real roots or no real roots at all. In order to calculate the real roots of a

given quadratic equation, we first calculate the **discriminant** (D) by the formula: D = b^2 - 4*ac*. If the discriminant is **zero**, then the quadratic equation has **one double real root** and it is calculated by the formula: $x_{1,2} = \frac{-b}{2a}$. If the value of the discriminant is **positive**, then the equation has **two distinct real roots**, which are calculated by the formula: $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. If the discriminant is **negative**, the quadratic equation has **no real roots**.

7. Use nested **if** statements. You could use the loop structure **for**, which you could read about in the "Loops" chapter of the book or in Internet.

8. First input a variable, which indicates **what type will be the input**, i.e. by entering 0 the type is **int**, by 1 is **double** and by 2 is **string**.

9. Use nested **if** statements or series of **31 comparisons**, in order to check all the sums of the 31 subsets of the given numbers (without the empty one). Note that the problem in general (with **N** numbers) is complex and using loops will not be enough to solve it.

10. Use **switch** statement or a sequence of **if**-**else** constructs and at the end print at the console the calculated points.

11. Use nested **switch** statements. Pay special attention to the numbers from 0 to 19 and those that end with 0. There are **many special cases!**

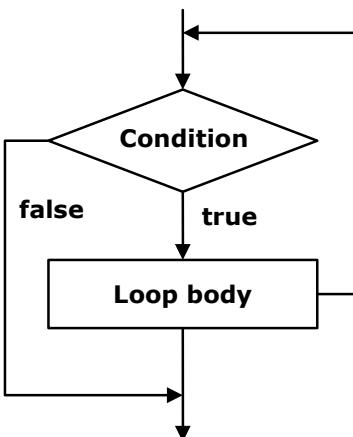
    You might benefit from using **methods** to reuse the code you write, because printing a single digit is part of printing a 2-digit number which is part of printing 3-digit number. You may read about methods in the chapter "Methods".

Chapter 6. Loops

In This Chapter

In this chapter we will examine the **loop programming constructs** through which we can execute a code snippet repeatedly. We will discuss how to implement conditional repetitions (**while and do-while loops**) and how to work with **for-loops**. We will give examples of different possibilities to define loops, how to construct them and some of their key usages. Finally, we will discuss the **foreach-loop** construct and how we can use multiple loops placed inside each other (**nested loops**).

What Is a "Loop"?

In programming often requires repeated execution of a sequence of operations. A **loop** is a basic programming construct that allows **repeated execution of a fragment** of source code. Depending on the type of the loop, the code in it is repeated a fixed number of times or repeats until a given condition is true (exists).

Loops that never end are called **infinite loops**. Using an infinite loop is rarely needed except in cases where somewhere in the body of the loop a **break** operator is used to terminate its execution prematurely. We will cover this later but now let's look how to create a loop in the C# language.

While Loops

One of the simplest and most commonly used loops is **while**.

```
while (condition)
{
   loop body;
}
```

In the code above example, **condition** is any **expression that returns a Boolean result – true** or **false**. It determines how long the loop body will be repeated and is called the **loop condition**. In this example the **loop body** is the programming code executed at each iteration of the loop, i.e. whenever the input condition is true. The behavior of **while** loops can be represented by the following scheme:

In the **while** loop, first of all the Boolean expression is calculated and if it is **true** the sequence of operations in the body of the loop is executed. Then again the input condition is checked and if it is true again the body of the loop is executed. All this is repeated again and again **until at some point the conditional expression returns value false**. At this point the loop stops and the program continues to the next line, immediately after the body of the loop.

The body of the **while** loop may not be executed even once if in the beginning the condition of the cycle returns **false**. If the condition of the cycle is never broken the loop will be executed indefinitely.

Usage of While Loops

Let's consider a very simple example of using the **while loop**. The purpose of the loop is to print on the console the numbers in the range from 0 to 9 in ascending order:

```
// Initialize the counter
int counter = 0;
// Execute the loop body while the loop condition holds
while (counter <= 9)
{
   // Print the counter value
   Console.WriteLine("Number : " + counter);
   // Increment the counter
   counter++;
}
```

When executing the sample code we obtain the following result:

```
Number : 0
```

```
Number : 1
Number : 2
Number : 3
Number : 4
Number : 5
Number : 6
Number : 7
Number : 8
Number : 9
```

Let's give some more examples in order to illustrate the usefulness of loops and to show some problems that can be solved by using loops.

Summing the Numbers from 1 to N

In this example we will examine how by using the **while loop** we can find the sum of the numbers from **1** to **n**. The number **n** is read from the console:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
int num = 1;
int sum = 1;
Console.Write("The sum 1");
while (num < n)
{
   num++;
   sum += num;
   Console.Write(" + " + num);
}
Console.WriteLine(" = " + sum);
```

First we initialize the variables **num** and **sum** with the value of **1**. In **num** we keep the current number, which we add to the sum of the preceding numbers. Trough each loop we increase **num** with 1 to get the next number, then in the condition of the loop we check whether it is in the range from **1** to **n**. The **sum** variable contains the sum of the numbers from 1 to **num** at any time. Upon entering the loop we add to **sum** the next number stored in **num**. We print on the console all **num** numbers from 1 to n with a separator "**+**" and the final result of the summing after the loop's ending. The result of the program's execution is as follows (we enter n = 17):

```
N = 17
The sum 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 +
14 + 15 + 16 + 17 = 153
```

Let's give another example of using the **while** loop, before moving on to other structures for organizing loops.

Check If a Number Is Prime – Example

We will write a **program to check whether a given number is prime or not**. We will read the number to check from the console. As we know from the mathematics, a prime number is any positive integer number, which, is not divisible by any other numbers except 1 and itself. We can check if the number **num** is prime when in a loop we check if it divides by numbers from 2 to **√num**:

```
Console.Write("Enter a positive number: ");
int num = int.Parse(Console.ReadLine());
int divider = 2;
int maxDivider = (int)Math.Sqrt(num);
bool prime = true;
while (prime && (divider <= maxDivider))
{
   if (num % divider == 0)
   {
      prime = false;
   }
   divider++;
}
Console.WriteLine("Prime? " + prime);
```

We use the variable **divider** to store the value of a potential divisor of the number. First we initialize it with 2 (the smallest possible divider). The variable **maxDivider** is the maximum possible divisor, which is equal to the square root of the number. If we have a divisor bigger than **√num**, then **num** should also have another divisor smaller than **√num** and that's why it's useless to check the numbers bigger than **√num**. This way we reduce the number of loop iterations.

For the result we use a Boolean variable called **prime**. Initially, its value is **true**. While passing through the loop, if it turns out that the number has a divisor, the value of **prime** will become **false**.

The condition of the **while** loop consists of two other sub-conditions which are related to the logical operator (logical **and**). In order to execute the loop, these two sub-conditions must be true simultaneously. If at some point we find a divisor of the number **num**, the variable **prime** becomes **false** and the condition of the loop is no longer satisfied. This means that the loop is executed until it finds the first divisor of the number or until it proves the fact that **num** is not divisible by any of the numbers in the range from 2 to **√num**.

Here is how the result of the above example's execution looks like if the input values are respectively the numbers 37 and 34:

```
Enter a positive number: 37
Prime? True
```

```
Enter a positive number: 34
Prime? False
```

Operator "break"

The **break** operator is used for prematurely **exiting the loop**, before it has completed its execution in a natural way. When the loop reaches the **break** operator it is terminated and the program's execution continues from the line immediately after the loop's body. A loop's termination with the **break** operator can only be done from its body, during an iteration of the loop. When **break** is executed the code in the loop's body after it is skipped and not executed. We will demonstrate exiting from loop with **break** with an example.

Calculating Factorial – Example

In this example we will calculate the factorial of a number entered from the console. The calculation is performed by using an **infinite while loop** and the **operator break**. Let's remember from the mathematics what is factorial and how it is calculated. The factorial of an integer n is a function that is calculated as a product of all integers less than or equal to n or equal to it. It is written down as n! and by definition the following formulas are valid for it:

- N! = 1 * 2 * 3 … (n-1) * n, for n> 1;
- 2! = 1 * 2;
- 1! = 1;
- 0! = 1.

The product n! can be expressed by a factorial of integers less than n:

- N! = (N-1)! * N, by using the initial value of 0! = 1.

In order to calculate the factorial of n we will directly use the definition:

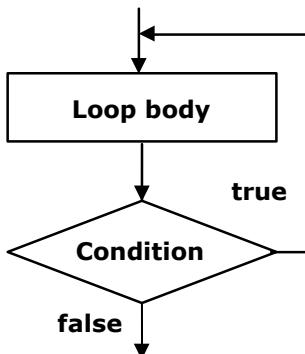
```
int n = int.Parse(Console.ReadLine());
// "decimal" is the biggest C# type that can hold integer values
decimal factorial = 1;
// Perform an "infinite loop"
while (true)
{
  if (n <= 1)
  {
```

```
        break;
    }
    factorial *= n;
    n--;
}
Console.WriteLine("n! = " + factorial);
```

First we initialize the variable **factorial** with 1 and read n from the console. We construct an endless **while** loop by using **true** as a condition of the loop. We use the **break** operator, in order to terminate the loop, when n reaches a value less than or equal to 1. Otherwise, we multiply the current result by n and we reduce n with one unit. Practically in the first iteration of the loop the variable **factorial** has a value n, in the second – n*(n-1) and so on. In the last iteration of the loop the value of **factorial** is the product n*(n-1)*(n-2)*…*3*2, which is the desired value of n!.

If we execute the sample program and enter 10 as input, we obtain the following result:

```
10
n! = 3628800
```

Do-While Loops

The **do-while** loop is similar to the **while** loop, but it checks the condition after each execution of its loop body. This type of loops is called loops with condition at the end (post-test loop). A **do-while** loop looks like this:

```
do
{
    executable code;
} while (condition);
```

By design **do-while** loops are executed according to the following scheme:

Initially the **loop body** is executed. Then its **condition is checked**. If it is true, the loop's body is repeated, otherwise the loop ends. This logic is repeated until **the condition of the loop is broken**. The body of the loop is executed at least once. If the loop's condition is constantly true, the loop never ends.

Usage of Do-While Loops

The **do-while** loop is used when we want to guarantee that the sequence of operations in it will be executed repeatedly and at least once in the beginning of the loop.

Calculating Factorial – Example

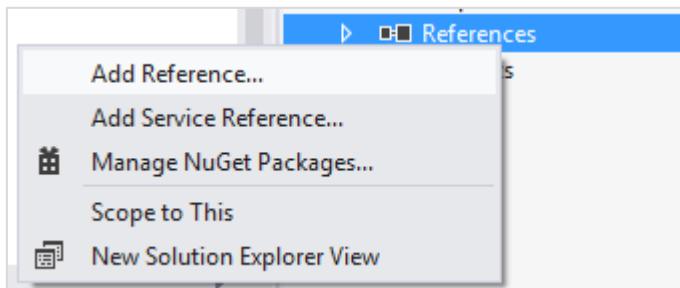
In this example we will again calculate the factorial of a given number n, but this time instead of an infinite **while** loop we will use a **do-while**. The logic is similar to that in the previous example:

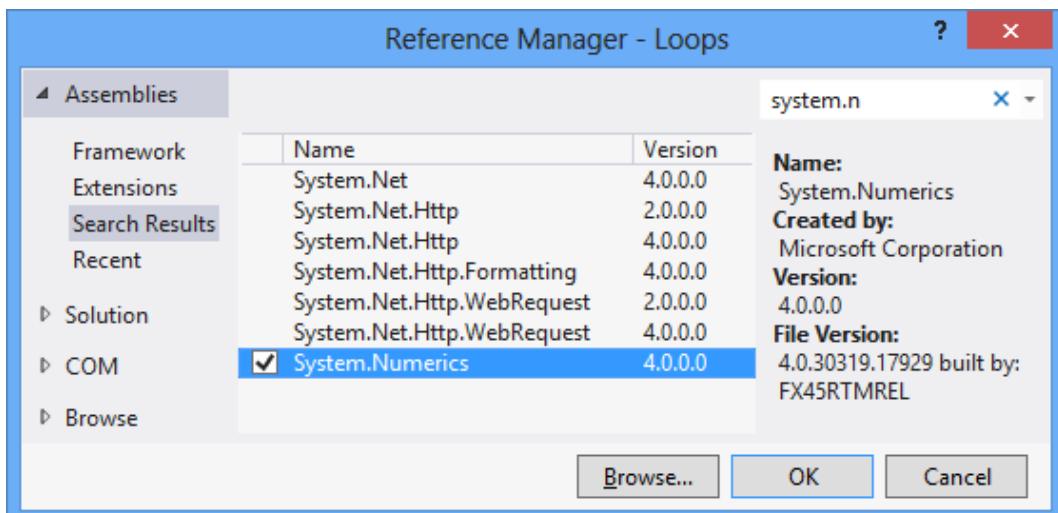
```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
decimal factorial = 1;
do
{
  factorial *= n;
  n--;
} while (n > 0);
Console.WriteLine("n! = " + factorial);
```

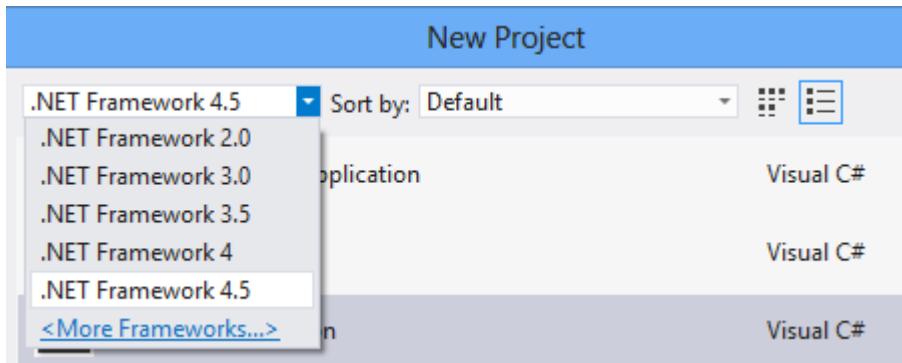
At the beginning we start with a result of 1 and multiply consecutively the result at each iteration by n, and reduce n by one unit, until n reaches 0. This gives us the product n*(n-1)*…*1. Finally, we print the result on the console. This algorithm always performs at least one multiplication and that's why it will not work properly when n ≤ 0.

Here is the result of the above example's execution for n=7:

```
n = 7
n! = 5040
```

Factorial of a Large Number – Example

You might be wondering what will happen if we set a large value for the number n in the previous example, say n=100. Then when, calculating the n! we will overflow the **decimal** type and the result will be an exception of type **System.OverflowException**:

```
n = 100
```

```
Unhandled Exception: System.OverflowException: Value was either
too large or too small for a Decimal.
   at System.Decimal.FCallMultiply(Decimal& result, Decimal d1,
Decimal d2)
   at System.Decimal.op_Multiply(Decimal d1, Decimal d2)
   at TestProject.Program.Main() in
C:\Projects\TestProject\Program
.cs:line 17
```

If we want to calculate 100! we can use data type **`BigInteger`** (which is new as of .NET Framework 4.0 and is missing in the older .NET versions). This type represents an integer, which can be **very large** (for example 100,000 digits). There is no limit on the size of the numbers recorded in the class **`BigInteger`** (as long as you have enough RAM).

In order to use **`BigInteger`**, we must first **add a reference** from our project to the assembly **`System.Numerics.dll`** (this is a standard .NET library for working with very large integers, which is not referenced by default by our VS projects). **Adding a reference** to it is done by right-clicking on the current project references in the Solution Explorer window of Visual Studio:

We search and choose **the assembly `System.Numerics.dll`** from the list:

If the assembly is missing from the list, that means that the Visual Studio project probably does not target **.NET Framework 4.0** or above and you must either create a new project or change the version of the current one:

Then we need to add "**using System.Numerics;**" before the beginning of the class of our program and replace **decimal** with **BigInteger**. The program obtains the following form:

```
using System;
using System.Numerics;

class Factorial
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());
      BigInteger factorial = 1;
      do
      {
         factorial *= n;
         n--;
      } while (n > 0);
      Console.WriteLine("n! = " + factorial);
   }
}
```

If we now run the program for n=100, we will get the value of **100 factorial**, which is a 158-digit number:

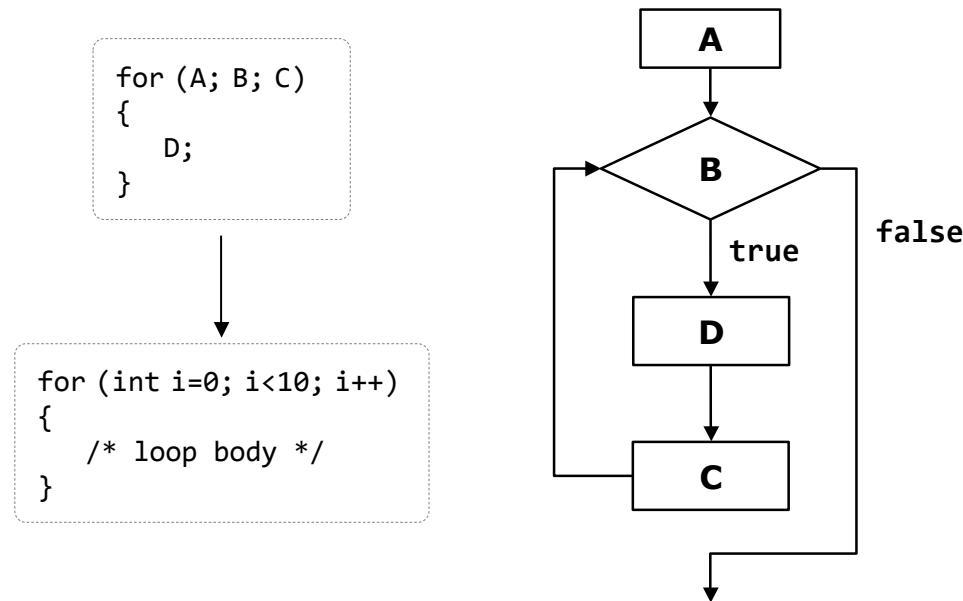
```
n = 100
n! =
9332621544394415268169923885626670049071596826438162146859296389
5217599993229915608941463976156518286253697920827223758251185210
916864000000000000000000000000
```

By **BigInteger** you can calculate 1000!, 10000! and even 100000! It will take some time, but **OverflowException** will not occur. The **BigInteger** class is very powerful but it works many times slower than **int** and **long**. For our unpleasant surprise there is no class "big decimal" in .NET Framework, only "big integer".

Product in the Range [N…M] – Example

Let's give another, more interesting example of working with **do-while** loops. The goal is to find the product of all numbers in the range [n…m]. Here is an example solution to this problem:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());

Console.Write("m = ");
int m = int.Parse(Console.ReadLine());

int num = n;
long product = 1;
do
{
  product *= num;
  num++;
} while (num <= m);

Console.WriteLine("product[n...m] = " + product);
```

In the example code we consecutively assign to **num** at each iteration the values n, n+1, …, m and in the variable **product** we accumulate the product of these values. We require the user to enter n, which should be less than m. Otherwise we will receive as a result the number n.

If we run the program for n=2 and m=6 we will obtain the following result:

```
n = 2
m = 6
product[n...m] = 720
```

Be careful: the product grows very fast, so you may need to use **BigInteger** instead of **long** for the calculated result. Also beware of hidden integer overflow. Unchecked code will silently overflow and the code above will produce incorrect output instead of showing an error. To overcome this, you may surround the line holding the multiplication by the [**checked** keyword](#).

For Loops

For-loops are a slightly more complicated than **while** and **do-while** loops but on the other hand they can solve more complicated tasks with less code. Here is the scheme describing **for**-loops:

They contain an **initialization block** (A), **condition** (B), **body** (D) and **updating commands** for the loop variables (C). We will explain them in details shortly. Before that, let's look at how the program code of a **for**-loop looks like:

```
for (initialization; condition; update)
{
  loop's body;
}
```

It consists of an **initialization part** for the counter (in the pattern **int i = 0**), a Boolean condition (**i < 10**), an expression for updating the counter (**i++**, it might be **i--** or for instance, **i = i + 3**) and body of the loop.

The **counter of the loop** distinguishes it from other types of loops. Most often the counter changes from a given initial value to a final one in ascending order, for example from 1 to 100. The number of iterations of a given **for**-loop is usually known before its execution starts. A **for**-loop can have one or several loop variables that move in ascending or descending order or with a step. It is possible one loop variable to increase and the other – to decrease. It is even possible to make a loop from 2 to 1024 in steps of multiplication by 2, since the update of the loop variables can contain not only addition, but any other arithmetic (as well as other) operations.

Since none of the listed elements of the **for**-loops is mandatory, we can skip them all and we will get an **infinite loop**:

```
for ( ; ; )
{
  // Loop body
}
```

Now let's consider in details the separate parts of a **for**-loop.

Initialization of For Loops

For-loops can have an **initialization block**:

```
for (int num = 0; …; …)
{
  // The variable num is visible here and it can be used
}
// Here num can not be used
```

It is **executed only once**, just before entering the loop. Usually the initialization block is used to declare the **counter-variable** (also called a **loop variable**) and to set its initial value. This variable is "visible" and can be used only within the loop. In the initialization block is possible to declare and initialize more than one variable.

Condition of the For Loop

For-loops can have a **loop condition**:

```
for (int num = 0; num < 10; …)
{
  // Loop body
}
```

The condition (**loop condition**) is evaluated once before each iteration of the loop, just like in the **while** loops. For result **true** the loop's body is executed, for result **false** it is skipped and the loop ends (the program continues immediately after the last line of the loop's body).

Update of the Loop Variables

The last element of a **for**-loop contains **code that updates** the loop variable:

```
for (int num = 0; num < 10; num++)
{
  // Loop body
```

```
}
```

This code is executed at each iteration, after the loop's body has been executed. It is most commonly used to update the value of the counter-variable.

The Body of the Loop

The **body of the loop** contains a block with source code. The loop variables, declared in the initialization block of the loop are available in it.

For-Loop – Example

Here is a complete example of a **for**-loop:

```
for (int i = 0; i <= 10; i++)
{
  Console.Write(i + " ");
}
```

The result of its execution is the following:

```
0 1 2 3 4 5 6 7 8 9 10
```

Here is another, more complicated example of a **for**-loop, in which we have two variables **i** and **sum**, that initially have the value of 1, but we update them consecutively at each iteration of the loop:

```
for (int i = 1, sum = 1; i <= 128; i = i * 2, sum += i)
{
  Console.WriteLine("i={0}, sum={1}", i, sum);
}
```

The result of this loop's execution is the following:

```
i=1, sum=1
i=2, sum=3
i=4, sum=7
i=8, sum=15
i=16, sum=31
i=32, sum=63
i=64, sum=127
i=128, sum=255
```

Calculating N^M – Example

As a further example we will write a program that raises the number n to a power of m, and for this purpose we will use a **for**-loop:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write("m = ");
int m = int.Parse(Console.ReadLine());
decimal result = 1;
for (int i = 0; i < m; i++)
{
  result *= n;
}
Console.WriteLine("n^m = " + result);
```

First we initialize the result (**result = 1**). The loop starts by setting an initial value for the counter-variable (**int i = 0**). We define the condition for the loop's execution (**i < m**). This way the loop will be executed from **0** to **m-1** i.e. exactly **m** times. During each run of the loop we multiply the result by **n** and so **n** will be raised to the next power (1, 2, …, **m**) at each iteration. Finally we print the result to see if the program works properly.

Here is how the outcome of the program for n = 2 and m = 10 looks like:

```
n = 2
m = 10
n^m = 1024
```

For-Loop with Several Variables

As we have already seen, in the construct of a **for**-loop we can use **multiple variables** at the same time. Here is an example in which we have two counters. One of the counters moves up from 1 and the other moves down from 10:

```
for (int small=1, large=10; small<large; small++, large--)
{
  Console.WriteLine(small + " " + large);
}
```

The condition for loop termination is overlapping of the counters. Finally we get the following result:

```
1 10
2 9
```

```
3 8
4 7
5 6
```

Operator "continue"

The **continue** operator **stops the current iteration** of the inner loop, without terminating the loop. With the following example we will examine how to use this operator.

We will calculate the sum of all odd integers in the range [1…n], which are not divisible by 7 by using the **for**-loop:

```
int n = int.Parse(Console.ReadLine());
int sum = 0;
for (int i = 1; i <= n; i += 2)
{
  if (i % 7 == 0)
  {
    continue;
  }
  sum += i;
}
Console.WriteLine("sum = " + sum);
```

First we initialize the **loop's variable** with a value of 1 as this is the first odd integer within the range [1…n]. After each iteration of the loop we check if **i** has not yet exceeded **n** (**i <= n**). In the expression for updating the variable we increase it by 2 in order to pass only through the odd numbers. Inside the loop body we check whether the current number is divisible by 7. If so we call the operator **continue**, which skips the rest of the loop's body (it skips adding the current number to the sum). If the number is not divisible by seven, it continues with updating of the sum with the current number.

The result of the example for n = 11 is as follows:

```
11
sum = 29
```

Foreach Loops

The **foreach loop** (extended **for**-loop) is new for the C/C++/C# family of languages, but is well known for the VB and PHP programmers. This programming construct serves to **iterate over all elements** of an array, list or other collection of elements (**IEnumerable**). It passes through all the elements of the specified collection even if the collection is not indexed.

We will discuss arrays in more details in chapter "Arrays", but for now we can imagine one array as an ordered sequence of numbers or other elements.

Here is how a **foreach** loop looks like:

```
foreach (type variable in collection)
{
  statements;
}
```

As we see, **it is significantly simpler than the standard for-loop** and therefore is very often preferred by developers because it saves writing when you need to go through all the elements of a given collection. Here is an example that shows how we can use **foreach**:

```
int[] numbers = { 2, 3, 5, 7, 11, 13, 17, 19 };
foreach (int i in numbers)
{
  Console.Write(" " + i);
}
Console.WriteLine();
string[] towns = { "London", "Paris", "Milan", "New York" };
foreach (string town in towns)
{
  Console.Write(" " + town);
}
```

In the example we create an array of numbers, which are after that went through with a **foreach** loop, and its elements are printed on the console. Then an array of city names (strings) is created and in the same way it is went through and its elements are printed on the console. The result of the example is:

```
2 3 5 7 11 13 17 19
London Paris Milan New York
```

Nested Loops

The **nested loops** are programming constructs consisting of several loops located into each other. The innermost loop is executed more times, and the outermost – less times. Let's see how two nested loops look like:

```
for (initialization, verification, update)
{
  for (initialization, verification, update)
  {
```

```
        executable code
    }
    …
}
```

After initialization of the first **for** loop, the execution of its body will start, which contains the second (nested) loop. Its variable will be initialized, its condition will be checked and the code within its body will be executed, then the variable will be updated and execution will continue until the condition returns **false**. After that the second iteration of the first **for** loop will continue, its variable will be updated and the whole second loop will be performed once again. The inner loop will be fully executed as many times as the body of the outer loop.

Let’s now consider a real example that will demonstrate how useful the nested loops are.

Printing a Triangle – Example

Let’s solve the following problem: for a given number n, to print on the console a triangle with n number of lines, looking like this:

```
1
1 2
1 2 3
…
1 2 3 … n
```

We will solve the problem with two **for**-loops. The **outer loop** will traverse the lines, and the inner one – the elements in them. When we are on the first line, we have to print "1" (1 element, 1 iteration of the inner loop). On the second line we have to print "1 2" (2 elements, 2 iterations of the internal loop). We see that there is a correlation between the line on which we are and the number of the elements that we print. This tells us how to organize the **inner loop’s structure**:

- We initialize the loop variable with 1 (the first number that we will print): **col** = 1;
- The repetition condition depends on the line on which we are: **col** <= **row**;
- We increase the loop variable with one unit at each iteration of the internal loop.

Basically, we need to implement a **for**-loop (external) from 1 to n (for the lines) and put another **for-**loop (internal) in it – for the numbers on the current line, which should spin from 1 to the number of the current line. The

external loop should go through the lines while the internal – through the columns of the current line.

Finally we get the following code:

```
int n = int.Parse(Console.ReadLine());
for (int row = 1; row <= n; row++)
{
   for (int col = 1; col <= row; col++)
   {
      Console.Write(col + " ");
   }
   Console.WriteLine();
}
```

If we execute it, we will make sure that it works correctly. Here is how the result for n=7 looks like:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

Note: when n > 9 the triangle will have a small defect. Think how to fix it!

Prime Numbers in an Interval – Example

Let's consider another **example of nested loops**. We set a goal to print on the console all prime number in the interval [n…m]. We will limit the interval by a **for**-loop and in order to check for a prime number we will use a nested **while** loop:

```
Console.Write("n = ");
int n = int.Parse(Console.ReadLine());
Console.Write("m = ");
int m = int.Parse(Console.ReadLine());

for (int num = n; num <= m; num++)
{
   bool prime = true;
   int divider = 2;
   int maxDivider = (int)Math.Sqrt(num);
   while (divider <= maxDivider)
```

```
  {
    if (num % divider == 0)
    {
      prime = false;
      break;
    }
    divider++;
  }
  if (prime)
  {
    Console.Write(" " + num);
  }
}
```

Using the outer **for**-loop we check each of the numbers n, n+1, …, m if it is prime. At each iteration of the outer **for**-loop we check whether its loop variable **num** is a prime number. The logic by which we check for a prime number is already familiar to us. At first we initialize the variable **prime** with **true**. With the internal **while** loop we check for each of the numbers [2…√**num**] if it is a divisor of **num** and if so, we set **prime** to **false**. After finishing the **while** loop the Boolean variable **prime** indicates whether the number is prime or not. If the number is prime we print it on the console.

If we execute the example for n=3 and m=75, we will obtain the following result:

```
n = 3
m = 75
 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
```

Lucky Numbers – Example

Let's consider another example through which we will show that we can put even **more than two loops into each other**. Our purpose is to find and print all four-digit numbers of the type ABCD, where: A+B = C+D (we call them lucky numbers). We will implement it with four **for**-loops – one for each digit. The outermost loop will define the thousands. It will start from 1 and the rest of the loops – from 0. They will determine the hundreds, the tens and the units. We will perform a check if our current number in the most inner loop is a lucky one and if so, we will print it on the console. Here is an implementation example:

```
for (int a = 1; a <= 9; a++)
{
  for (int b = 0; b <= 9; b++)
```

```
    {
      for (int c = 0; c <= 9; c++)
      {
        for (int d = 0; d <= 9; d++)
        {
          if ((a + b) == (c + d))
          {
            Console.WriteLine(
              " " + a + " " + b + " " + c + " " + d);
          }
        }
      }
    }
}
```

Here is a part of the printed result (the entire result is too long):

```
1 0 0 1
1 0 1 0
1 1 0 2
1 1 1 1
1 1 2 0
1 2 0 3
1 2 1 2
1 2 2 1
…
```

We leave as homework for the diligent reader to offer a more efficient solution to the same problem, using only three nested loops rather than four.

Lottery 6/49 – Example

In the following example we will find **all possible combinations of the lottery game "6/49"**. We have to find and print all possible extracts of 6 different numbers, each of which is in the range [1…49]. We will use 6 **for**-loops. Unlike the previous example, the numbers cannot be repeated. To avoid repetitions we will strive for each subsequent number to be larger than the previous. Therefore, the internal loops will not start from 1 but from the number to which the previous loop got + 1. We will have to go through the first loop until it reaches 44 (and not to 49), the second – 45, etc. The last loop will be up to 49. If you make all loops to 49 you will receive matching numbers in certain combinations. For the same reason, each subsequent cycle starts from the previous loop counter + 1. Let's see what will happen:

```
for (int i1 = 1; i1 <= 44; i1++)
{
```

```
    for (int i2 = i1 + 1; i2 <= 45; i2++)
    {
      for (int i3 = i2 + 1; i3 <= 46; i3++)
      {
        for (int i4 = i3 + 1; i4 <= 47; i4++)
        {
          for (int i5 = i4 + 1; i5 <= 48; i5++)
          {
            for (int i6 = i5 + 1; i6 <= 49; i6++)
            {
              Console.WriteLine(i1 + " " + i2 + " " +
                i3 + " " + i4 + " " + i5 + " " + i6);
            }
          }
        }
      }
    }
}
```

Everything looks correct. Let's run the program. It seems to work but there is one problem – there are too many combinations and the program does not end (it is **so slow**, that hardly anyone is going to wait). This is correct and it is one of the reasons why there is Lottery 6/49 – there really are lots of combinations. We are leaving the curious reader to practice changing the example above just to calculate all lottery combinations, instead of printing them. This change will dramatically reduce the volume of the printed results on the console and the program will end surprisingly quickly.

Printing excessive amount of text on the console is very slow and should be avoided. A modern computer (as of 2012) can perform 300,000,000 operations per second but can print only 10,000 – 20,000 text lines per second.

Exercises

1. Write a program that prints on the console **the numbers from 1 to N**. The number **N** should be read from the standard input.

2. Write a program that prints on the console the numbers from 1 to N, which are **not divisible by 3 and 7 simultaneously**. The number N should be read from the standard input.

3. Write a program that reads from the console a series of integers and prints the **smallest** and **largest** of them.

4. Write a program that prints **all possible cards from a standard deck** of cards, without jokers (there are 52 cards: 4 suits of 13 cards).

5. Write a program that reads from the console number N and print the sum of the first N members of the **Fibonacci sequence**: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …

6. Write a program that calculates **N!/K!** for given N and K (1<K<N).

7. Write a program that calculates **N!*K!/(N-K)!** for given N and K (1<K<N).

8. In combinatorics, the **Catalan numbers** are calculated by the following formula: $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$, for n ≥ 0. Write a program that calculates the n^{th} Catalan number by given n.

9. Write a program that for a given integers **n** and **x**, **calculates the sum**: $S = 1 + 1!/x + 2!/x^2 + \ldots + n!/x^n$

10. Write a program that reads from the console a **positive integer number N** (N < 20) and prints a **matrix** of numbers as on the figures below:

N = 3

1	2	3
2	3	4
3	4	5

N = 4

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

11. Write a program that calculates with **how many zeroes the factorial of a given number ends**. Examples:

    N = 10 -> N! = 36288**00** -> 2

    N = 20 -> N! = 243290200817664**0000** -> 4

12. Write a program that converts a given number **from decimal to binary notation** (numeral system).

13. Write a program that converts a given number **from binary to decimal notation**.

14. Write a program that converts a given number **from decimal to hexadecimal notation**.

15. Write a program that converts a given number **from hexadecimal to decimal notation**.

16. Write a program that by a given integer **N** prints the numbers from 1 to N in **random order**.

17. Write a program that given two numbers finds their **greatest common divisor (GCD)** and their **least common multiple (LCM)**. You may use the formula **LCM(a, b) = |a*b| / GCD(a, b)**.

18. * Write a program that for a given number n, outputs a matrix in the form of a **spiral**:

Example for n=4:

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

Solutions and Guidelines

1. Use a **for**-loop.

2. Use a **for**-loop and the operator **%** for finding **the remainder** in integer division. A number **num** is not divisible by 3 and 7 simultaneously exactly when (**num % (3*7) == 0**).

3. First **read the count** of numbers, for example in a variable **n**. Then **consequently enter n numbers** with one **for** loop. While entering each new number, save in two variables the **smallest** and the **largest** number until this moment. At the start initialize the smallest and the largest number with **Int32.MaxValue** and **Int32.MinValue** respectively.

4. **Number the cards** from 2 to 14 (these numbers will match the cards 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A). **Number the suits** from 1 to 4 (1 – club, 2 – diamond, 3 – heart and 4 – spades). Now you can use the **two nested loops** and print each of the cards with two **switch** statements.

5. **Fibonacci numbers** start from 0 and 1, each additional is obtained as the **sum of the previous two**. You can find the first **n** Fibonacci numbers with a **for**-loop from 1 to **n**, and at each iteration calculate the next number by using the previous two (which you will keep in two additional variables).

6. Multiply the **numbers from K+1 to N** (think why this is correct). You may check the properties of the **factorial function** in Wikipedia: http://en.wikipedia.org/wiki/Factorial.

7. **One solution** is to calculate separately each factorial and at the end to perform the respective operations with the results.

   Think how you can **optimize the calculations**, in order to not calculate too many factorials! In **fractions** of factorials there are many possibilities to **reduce the same factors** in the numerator and denominator. These optimizations will not only reduce the calculations and increase the performance but will **save you from overflows** in some situations. You might need to use arrays **num[0..N]** and **denum[0..n]** to hold the factors in the numerator and in the denominator and to **cancel the fraction**. You may read about arrays in the chapter "Arrays".

8. Use the same concept of **canceling the faction of simple factors**, like you probably did in the **previous problem**.

   You may also read more about the **Catalan numbers** in Wikipedia (http://en.wikipedia.org/wiki/Catalan_number) and use the **recurrent formula** for calculating them.

9. You can solve the problem with a **for**-loop for **k**=0…**n**, by using three additional variables **factorial**, **power** and **sum** in which you will keep for the k^{th} iteration of the loop respectively **k!**, x^k and the **sum of the first k members** of sequence. If your implementation is good, you should have only one loop and you should not use external functions to calculate factorials and to raise power.

10. You should use **two nested loops**, similar to the problem "Printing a Triangle". The outer loop must run from 1 to N, and the inner – from the value of the outer loop to the value of the outer loop + N - 1.

11. The **number of zeros at the end of n!** depends on how many times the number 10 is a divisor of the factorial. Because the product 1*2*3…*n has a greater number of divisors 2, than 5 and because 10 = 2 * 5, then the **number of zeros in n!** is exactly as many as the **multipliers with value 5 in the product 1 * 2 * 3 * … * n**. Because every fifth number is divisible by 5, and every 25^{th} number is divisible by 5 two times, etc., the number of zeros in n! is **the sum: n/5 + n/25 + n/125 + …**

12. Read in Wikipedia what **numeral systems** are: http://en.wikipedia.org/wiki/Numeral_system. After that consider how you can **convert numbers from decimal numeral system to another**. Think about the opposite – moving from another numeral system to decimal. If you have difficulty, see the chapter "Numeral Systems".

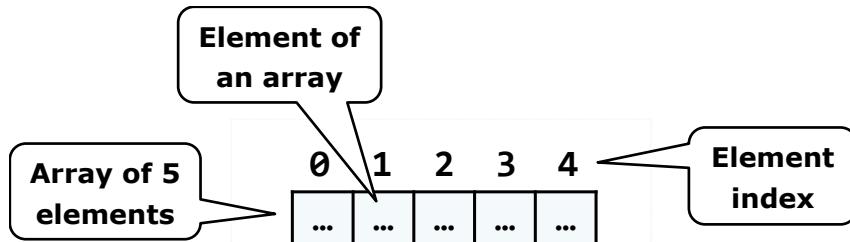
13. See the previous problem.

14. See the previous problem.

15. See the previous problem.

16. Search in the Internet for information about **the class System.Random**. Read in the Internet about **arrays** (or in the next chapter). Create an **array with N elements** and write in it the numbers from 1 to N. After that a few times (think exactly how many) **swap two random pairs** of elements from the array.

17. Search the Internet for the **Euclidean algorithm** for calculation the **greatest common divisor (CGD)** or read about it in Wikipedia: http://en.wikipedia.org/wiki/Euclidean_algorithm.

18. You should use a **two-dimensional array** (matrix). Search the Internet or see the chapter "Arrays". The algorithm of **filling a spiral matrix** in not straightforward and may require a bit of thinking. You might find helpful the "Spiral Matrix" problem from chapter "Sample Programming Exam – Topic #3".

Chapter 7. Arrays

In This Chapter

In this chapter we will learn about arrays as a way to **work with sequences of elements** of the same type. We will explain what **arrays** are, how we declare, create, instantiate and use them. We will examine one-dimensional and **multidimensional arrays**. We will learn different ways to iterate through the array, read from the standard input and write to the standard output. We will give many example exercises, which can be solved using arrays and we will show how useful they really are.

What Is an "Array"?

Arrays are vital for most programming languages. They are collections of variables, which we call **elements:**

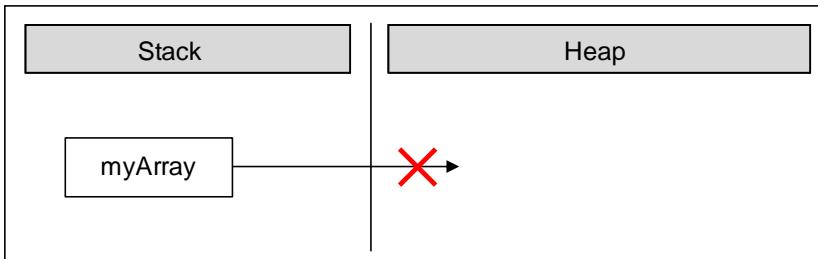
An array's elements in C# are numbered with 0, 1, 2, … N-1. Those numbers are called **indices**. The total number of elements in a given array we call **length of an array**.

All elements of a given array are of the same type, no matter whether they are **primitive** or **reference** types. This allows us to represent a group of similar elements as an ordered sequence and work on them as a whole.

Arrays can be in different dimensions, but the most used are the **one-dimensional** and the **two-dimensional** arrays. One-dimensional arrays are also called **vectors** and two-dimensional are also known as **matrices**.

Declaration and Allocation of Memory for Arrays

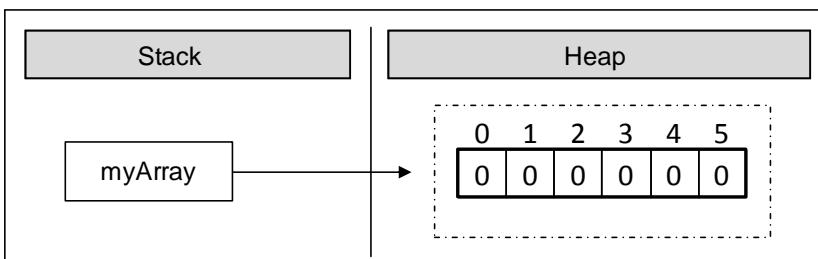
In C# the arrays have fixed length, which is set at the time of their instantiation and determines the total number of elements. Once the length of an array is set we cannot change it anymore.

Declaring an Array

We declare an array in C# in the following way:

```
int[] myArray;
```

In this example the variable **myArray** is the name of the array, which is of integer type (**int[]**). This means that we declared an array of integer numbers. With **[]** we indicate, that the variable, which we are declaring, is an array of elements, not a single element.

When we declare an array type variable, it is a **reference**, which does not have a value (it points to **null**). This is because the memory for the elements is not allocated yet.

The figure below shows how a declared array variable looks, when the memory for elements of the array is not allocated yet:

In the program's execution stack the variable with the name **myArray** is created and its value is set to **null** (meaning it holds no value).

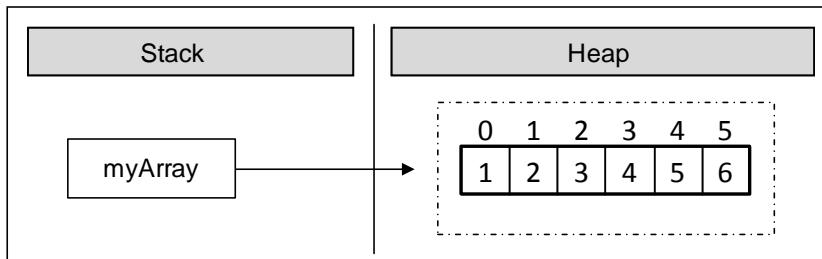
Creation of an Array – the Operator "new"

In C# we create an array with the help of the keyword **new**, which is used to allocate memory:

```
int[] myArray = new int[6];
```

In this example we allocate an array with length of 6 elements of type **int**. This means that in the dynamic memory (heap) an area of 6 integer numbers is allocated and they all are initialized with the value 0:

The figure shows, that after the allocation of memory for the array the variable **myArray** points to an address in the dynamic memory, where the values are. In C#, the elements of an array are always stored in the dynamic memory (called also **heap**).

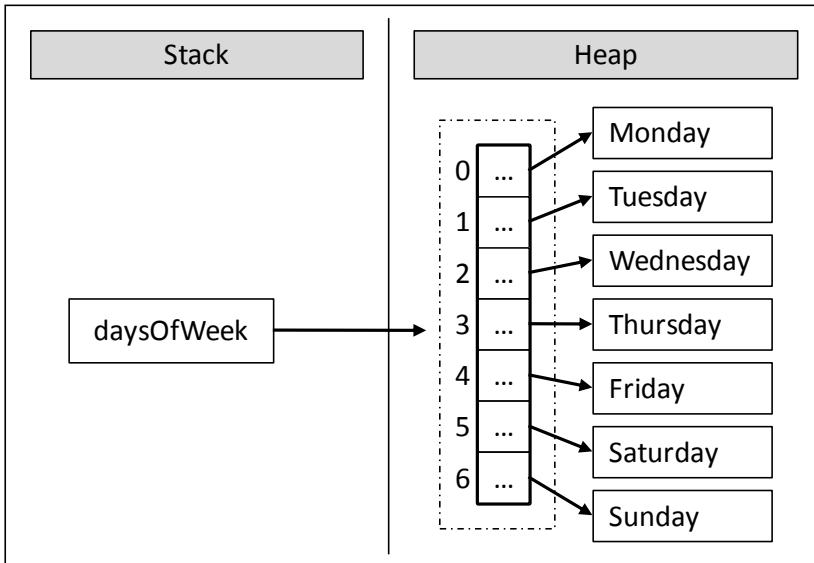
During the allocation of the memory for an array we set the total number of the elements in the brackets (a non-negative integer number), defining its length. The type of the elements is written after the reserved word **new**, so we indicate what type of elements are going to be allocated in the memory.

Array Initialization and Default Values

Before we can use an element of a given array, it has to be **initialized** or to have a **default value**. In some programming languages there are no default values and then if we try to access an element, which is not initialized, this may cause an error. In C# all variables, including the elements of arrays have a **default initial value**. This value is either **0** for the numeral types or its equivalent for the non-primitive types (for example **null** for a reference type and **false** for the **bool** type).

Of course we can set initial values explicitly. We can do this in different ways. Here is one of them:

```
int[] myArray = { 1, 2, 3, 4, 5, 6 };
```

In this case we create and initialize the elements of the array at the time of the declaration. On the figure below we see how the array is allocated in the memory when its values are initialized at the moment of its declaration:

With this syntax we use curly brackets instead of the operator **new**. Between the brackets we list the initial values of the array, separated by commas. Their count defines the length of the array.

Declaration and Initialization of an Array – Example

Here is one more example how to **declare and initialize an array**:

```
string[] daysOfWeek =
  { "Monday", "Tuesday", "Wednesday","Thursday", "Friday",
  "Saturday", "Sunday" };
```

In this case we allocate an array of seven elements of type **`string`**. The type **`string`** is a reference type (object) and its values are stored in the dynamic memory. The variable **`daysOfWeek`** is allocated in the stack memory, and points to a section of the dynamic memory containing the elements of the array. The type of each of these seven elements is **`string`**, which itself points to a different section of the dynamic memory, where the real value is stored.

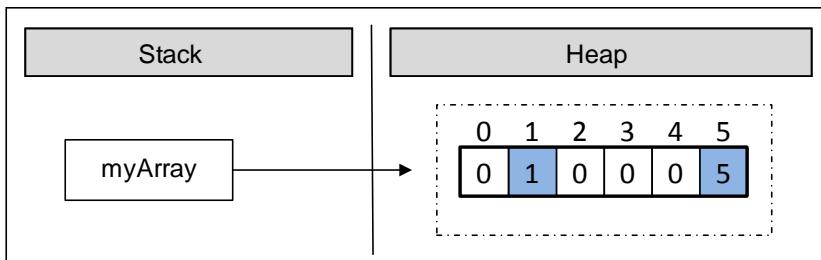
On this figure we see **how the array is allocated in the memory**:

Boundaries of an Array

Arrays are by default **zero-based**, which means the enumeration of the elements starts from **0**. The first element has the index 0, the second – 1, etc. In an array of **N** elements, the last element has the index **N-1**.

Access to the Elements of an Array

We access the array elements directly using their **indices**. Each element can be accessed through the name of the array and the element's **index** (consecutive number) placed in the brackets. We can access given elements of the array both for reading and for writing, which means we can treat elements as variables.

Here is an example for accessing an element of an array:

```
myArray[index] = 100;
```

In the example above we set a value of 100 to the element, which is at position **`index`**.

Here is an example, where we allocate an array of numbers and then we change some of them:

```
int[] myArray = new int[6];
myArray[1] = 1;
myArray[5] = 5;
```

After the change, the array is allocated in the memory as shown below:

As we can see, all elements, except those for which values are explicitly set, are initialized with the value 0 when the memory of the array was allocated.

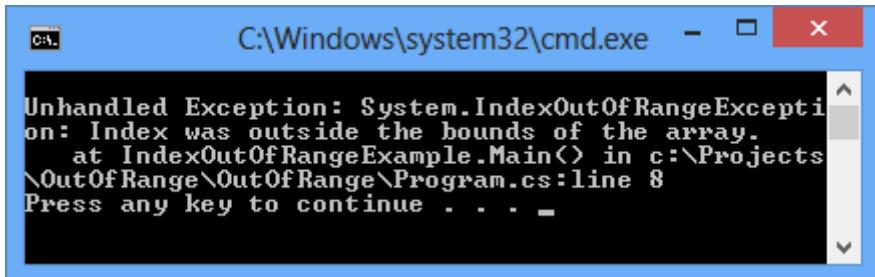
We can **iterate** through the array using a **loop** statement. The most common form of such iteration is by using a **for**-loop:

```
int[] arr = new int[5];
for (int i = 0; i < arr.Length; i++)
{
   arr[i] = i;
}
```

Going Out of Bounds of the Array

The .NET Framework does an **automatic check** on each element access attempt, whether the index is valid or it is **out of the range** of the array. When we try to access an invalid (not existing) element in an array, a **System.IndexOutOfRangeException** is thrown. The automatic check really helps the developers find errors while working with arrays. Of course, checking for exceptions has its price. Checks affect the performance, but that's nothing compared to avoiding errors like "out of range", "access to unallocated memory", etc.

Here is an example, where we are trying to access an element, which is out of the range of the array:

IndexOutOfRangeExample.cs

```
class IndexOutOfRangeExample
{
```

```
    static void Main()
    {
        int[] myArray = { 1, 2, 3, 4, 5, 6 };
        Console.WriteLine(myArray[6]);
    }
}
```

In the example above we allocate an array, which contains six integer numbers. The first index is 0, and the last index is 5. We are trying to print to the console an element with index 6, but because there is no such element this leads to an exception:

Reversing an Array – Example

In the next example we will access elements and change them using their indices. The task is to print the elements in **reversed order**. We will reverse the elements of the array using a second, auxiliary array, where we will keep the elements of the first one, but in a reversed order. Note that the length of both arrays is the same and it stays unchanged after the first allocation:

ArrayReverseExample.cs

```
class ArrayReverseExample
{
    static void Main()
    {
        int[] array = { 1, 2, 3, 4, 5 };
        // Get array size
        int length = array.Length;
        // Declare and create the reversed array
        int[] reversed = new int[length];

        // Initialize the reversed array
        for (int index = 0; index < length; index++)
        {
            reversed[length - index - 1] = array[index];
        }
```

```
        // Print the reversed array
        for (int index = 0; index < length; index++)
        {
            Console.Write(reversed[index] + " ");
        }
    }
}
// Output: 5 4 3 2 1
```

The example works in the following way: initially we allocate a one-dimensional array of type **int** and we initialize it with the numbers from 1 to 5. After that we keep the length of the array in the variable **length**. Note that we are using the property **Length**, which returns the total count of the elements of the array. In C# each array has a length property.

After that we declare the array **reversed** with the same **length**, where we will keep elements of the original array, but in a reversed order.

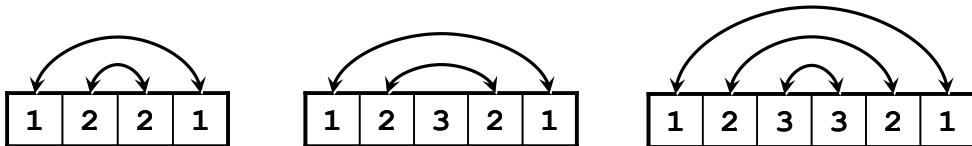
To reverse the elements we use a **for**-loop. At each iteration we increment the **index** variable by one and we make sure we access all consecutive elements of the **array**. The loop condition ensures that the array will be iterated from end to end.

Let's follow what happens when we iterate through the **array**. On the first iteration, **index** has a value of 0. Using **array[index]** we access the first element of the **array**, and respectively with **reversed[length - index - 1]** we access the last element of the new array **reversed** where we assign the values. Thus, we appropriated the value of the first element of the **array** to the last element of the **reversed** array. At each iteration **index** is incremented by one. This way, we access the next element in the order of **array** and the previous element in the order of **reversed**.

As a result we reversed the array and printed it. In the example we showed consecutive iterations through the array, which can also be done with different types of loop constructs (e.g. **while** and **foreach**).

Reading an Array from the Console

Let's see how we can read values of an array from the console. We will use a **for**-loop and the .NET Framework tools for reading from the console.

Initially we read a line from the console using **Console.ReadLine()**, and then we parse that line to an integer number using **int.Parse()** and we set it to the variable **n**. We then use the number **n** as length of the array.

```
int n = int.Parse(Console.ReadLine());
int[] array = new int[n];
```

Again we use a loop to iterate through the array. At each iteration we set the current element to what we have read from the console. The loop will continue **n** times, which means it will iterate through the array and so we will read a value for each element of the array:

```
for (int i = 0; i < n; i++)
{
  array[i] = int.Parse(Console.ReadLine());
}
```

Check for Symmetric Array – Example

An array is symmetric if the first and the last elements are equal and at the same time the second element and the last but one are equal as well and so on. On the figure a few examples for **symmetric arrays** are shown:

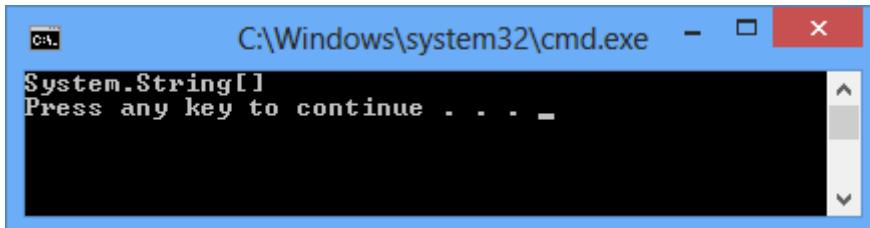
In the next example we will check whether an array is symmetric:

```
Console.Write("Enter a positive integer: ");
int n = int.Parse(Console.ReadLine());
int[] array = new int[n];

Console.WriteLine("Enter the values of the array:");

for (int i = 0; i < n; i++)
{
  array[i] = int.Parse(Console.ReadLine());
}

bool symmetric = true;
for (int i = 0; i < array.Length / 2; i++)
{
  if (array[i] != array[n - i - 1])
  {
    symmetric = false;
    break;
  }
}

Console.WriteLine("Is symmetric? {0}", symmetric);
```

We initialize an array and we read its elements from the console. We need to iterate through half of the array to check whether it is symmetric. The middle element of the array has an index **array.Length / 2**. If the length is an odd number this index is exactly the middle one, but if it is an even number, the index is to the right of the middle (the middle is between two elements). Thus the loop runs from **0** to **array.Length / 2** (non-inclusive).

To check whether an array is **symmetric**, we use a **bool** variable, and initially assume that the array is symmetric. During the iteration through the array we compare the first with the last element, the second with the last but one and so on. If at some point the compared elements are not equal, then we set the **bool** variable to **false**, which means the array is not symmetric.

In the end we print the value of the **bool** variable to the console.

Printing an Array to the Console

Often we have to **print the elements of a given array** to the console, after we have finished working with it.

We print elements of an array to the console similarly to the initializing of the elements, i.e. by using a loop to iterate through the array. There are no strict rules for printing, but often some sort of suitable formatting is used.

A frequent mistake is an attempt to print an array like a number:

```
string[] array = { "one", "two", "three", "four" };
Console.WriteLine(array);
```

Unfortunately this code **does not print the elements of an array**, just its type. Here is what we get after the execution of this code:

```
C:\Windows\system32\cmd.exe
System.String[]
Press any key to continue . . .
```

We print the elements of an array by hand, by using a **for**-loop:

```
string[] array = { "one", "two", "three", "four" };

for (int index = 0; index < array.Length; index++)
{
   // Print each element on a separate line
   Console.WriteLine("Element[{0}] = {1}", index, array[index]);
}
```

We are iterating through the array using the **for**-loop, which will go **array.Length** times, and we will print the current element using **Console.WriteLine()** and a formatted string. Here is the result:

```
Element[0] = one
Element[1] = two
Element[2] = three
Element[3] = four
```

Iteration through Elements of an Array

As we can see, the iteration through the elements of an array is one of the most used techniques when we work with arrays. **Consecutive iterating** using a loop will allow us to access each element through its index and we will be able to modify it as we want. We can do that with different loop constructs, but the most appropriate loop is the **for**-statement. We will examine in details how this type of iteration works.

Iteration with a For Loop

It is a good practice to use **for**-loops, when we work with arrays and structures with indices. In the following example we will double the values of all elements of an array of numbers and we will print them:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

Console.Write("Output: ");
for (int index = 0; index < array.Length; index++)
{
   // Doubling the number
   array[index] = 2 * array[index];
   // Print the number
   Console.Write(array[index] + " ");
}
// Output: 2 4 6 8 10
```

Using a **for**-loop we keep track of the current **index** of the array and we access the elements as needed. We do not have to iterate consecutively through all of them, which means the index that we are using in the **for**-loop may **iterate through the elements** in a way that our algorithm requires. For example we can iterate through some of the elements of the array, not through all of them:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

Console.Write("Output: ");
```

```
for (int index = 0; index < array.Length; index += 2)
{
  array[index] = array[index] * array[index];
  Console.Write(array[index] + " ");
}
// Output: 1 9 25
```

In this example we are iterating through all elements at even positions and we square their values.

Sometimes we want to **iterate through the array in a reverse order**. We do that in a similar way, except that the **for**-loop will start with the index of the last element and the index will decrease on each step until its value gets to 0 (inclusive). Here is an example:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

Console.Write("Reversed: ");
for (int index = array.Length - 1; index >= 0; index--)
{
  Console.Write(array[index] + " ");
}
// Reversed: 5 4 3 2 1
```

In this example we are iterating through the array in reverse order and we print each element to the console.

Iteration with "foreach" Loop

One of the most used constructs for iterating through elements of an array is **foreach**. The **foreach-loop construct in C#** is as follows:

```
foreach (var item in collection)
{
  // Process the value here
}
```

In this programming construct **var** is the type of the elements, which we iterate through. The **collection** is the array (or any other collection of elements) and **item** is the current element of the array on each step.

In general the **foreach** loop construct has the same properties like the **for**-loop. The main difference is that the iteration is made **always through all elements** – from the start to the end. We cannot access the current index, we are just iterating through the collection in a way, defined by the collection itself. For arrays the order of iteration is consecutive from the first element to

the last one. The loop variable in **foreach**-loops is **read-only** so we cannot modify the current loop item from the loop body.

The **foreach**-loop statement is used, when we do not need to change the elements, but just to read them.

Iteration with "foreach" Loop – Example

In the next example we will learn how to use the **foreach** loop to **iterate through the array**:

```
string[] capitals =
  { "Sofia", "Washington", "London", "Paris" };

foreach (string capital in capitals)
{
  Console.WriteLine(capital);
}
```

After we declared an array of strings **capitals**, we iterate through the array using **foreach** loop and we print the elements to the console. The current element on each step is stored in a variable **capital**. We get the following result when we execute the code:

```
Sofia
Washington
London
Paris
```

Multidimensional Arrays

The **one-dimensional arrays** are known also as **vectors** in mathematics. Often we need arrays with more than one dimension. For example we can easily represent the standard chess board as a two-dimensional array with size 8 by 8 (8 cells in a horizontal direction and 8 cells in a vertical direction).

What Is a Multidimensional Array? What Are Matrices?

Every valid type in C# can be used for a type of an array. So, we can have an array of arrays, which we will discuss later.

We declare a one-dimensional array of integer numbers using **int[]**, and we declare a two-dimensional with **int[,]**. This example shows that:

```
int[,] twoDimensionalArray;
```

Those arrays we will call **two-dimensional**, because they have two dimensions. They are also known as **matrices** (it is mathematical term). In general arrays with more than one dimension we will call **multidimensional**.

This way we can declare three-dimensional arrays as we add one more dimension:

```
int[,,] threeDimensionalArray;
```

In theory there is **no limit for an array dimensions**, but in practice we do not use much arrays with more than two dimensions therefore we will focus on two-dimensional arrays.

Multidimensional Array Declaration and Allocation

We declare multidimensional arrays in a way similar to one-dimensional arrays. Each dimension except the first is marked with comma:

```
int[,] intMatrix;
float[,] floatMatrix;
string[,,] strCube;
```

In the example above we create **two-dimensional** and **three-dimensional** arrays. Each dimension is represented by a comma in the square brackets **[]**.

We are allocating memory for multidimensional arrays by using the **keyword new** and for each dimension we set a length in the brackets as shown:

```
int[,] intMatrix = new int[3, 4];
float[,] floatMatrix = new float[8, 2];
string[,,] stringCube = new string[5, 5, 5];
```

In this example **intMatrix** is a two-dimensional array with 3 elements of type **int[]** and each of those 3 elements has a length of 4. Two-dimensional arrays are difficult to understand explained that way. Therefore we can imagine them as **two-dimensional matrices**, which have rows and columns for the dimensions:

	0	1	2	3
0	1	3	6	2
1	8	5	9	1
2	4	7	3	0

The rows and the columns of the square matrices are numbered with indices from 0 to n-1. If a two-dimensional array has a size of **m** by **n**, there are exactly **m*n** elements.

Two-Dimensional Array Initialization

We initialize two-dimensional arrays in the same way as we initialize one-dimensional arrays. We can list the element values straight after the declaration:

```
int[,] matrix =
{
  {1, 2, 3, 4}, // row 0 values
  {5, 6, 7, 8}, // row 1 values
};
// The matrix size is 2 x 4 (2 rows, 4 cols)
```

In the example above we **initialize a two-dimensional array** of type integer with size of 2 rows and 4 columns. In the outer brackets we place the elements of the first dimension, i.e. the rows of the array. Each row contains one dimensional array, which we know how to initialize.

Accessing the Elements of a Multidimensional Array

Matrices have two dimensions and respectively we access each element by using two indices: one for the rows and one for the columns. Multidimensional arrays have **different indices for each dimension**.

Each dimension in a multidimensional array starts at index 0.

Let's examine the next example:

```
int[,] matrix =
{
  {1, 2, 3, 4},
  {5, 6, 7, 8},
};
```

The array **matrix** has 8 elements, stored in 2 rows and 4 columns. Each element can be accessed in the following way:

```
matrix[0, 0]  matrix[0, 1]  matrix[0, 2]  matrix[0, 3]
matrix[1, 0]  matrix[1, 1]  matrix[1, 2]  matrix[1, 3]
```

In this example we can access each element using indices. If we assign the index for rows to **row**, and the index for columns to **col**, then we can access any element as shown:

```
matrix[row, col]
```

When we use multidimensional arrays each element is unique and can be identified with indices from the array:

```
nDimensionalArray[index1, …, indexN]
```

Length of Multidimensional Arrays

Each dimension of a multidimensional array has its own **length**, which can be accessed during the execution of the program. Let's look at an example for a two-dimensional array:

```
int[,] matrix =
{
  {1, 2, 3, 4},
  {5, 6, 7, 8},
};
```

We can get the number of the rows of this two-dimensional array by using **matrix.GetLength(0)** and the number of all columns per row with **matrix.GetLength(1)**. So, in this case **matrix.GetLength(0)** returns 2 and **matrix.GetLength(1)** returns 4.

Printing Matrices – Example

In the next example we will demonstrate how we can print two-dimensional arrays to the console:

```
// Declare and initialize a matrix of size 2 x 4
int[,] matrix =
{
  {1, 2, 3, 4}, // row 0 values
  {5, 6, 7, 8}, // row 1 value
};

// Print the matrix on the console
for (int row = 0; row < matrix.GetLength(0); row++)
{
  for (int col = 0; col < matrix.GetLength(1); col++)
  {
    Console.Write(matrix[row, col]);
  }
  Console.WriteLine();
}
```

First we declare and initialize an array, which we want to iterate through and print to the console. The array is two-dimensional, therefore we use a **for-**

loop which will iterate through the rows and a nested for loop which **for each row will iterate through the columns**. At each iteration we will print the current element using the appropriate method to access this element by using its two indices (row and column). Finally, if we execute this piece of code we will get the following result:

```
1 2 3 4
5 6 7 8
```

Reading Matrices from the Console – Example

In this example we will learn how to **read a two-dimensional array** from the console. First, we read the values (lengths) of the two-dimensions and then by using two nested loops we assign the value of each element (and in the end we print out the values of the array):

```
Console.Write("Enter the number of the rows: ");
int rows = int.Parse(Console.ReadLine());

Console.Write("Enter the number of the columns: ");
int cols = int.Parse(Console.ReadLine());

int[,] matrix = new int[rows, cols];

Console.WriteLine("Enter the cells of the matrix:");

for (int row = 0; row < rows; row++)
{
   for (int col = 0; col < cols; col++)
   {
      Console.Write("matrix[{0},{1}] = ",row, col);
      matrix[row, col] = int.Parse(Console.ReadLine());
   }
}

for (int row = 0; row < matrix.GetLength(0); row++)
{
   for (int col = 0; col < matrix.GetLength(1); col++)
   {
      Console.Write(" " + matrix[row, col]);
   }
   Console.WriteLine();
}
```

The program output when we execute it (in this case the array consists of three rows and two columns) is:

```
Enter the number of the rows: 3
Enter the number of the columns: 2
Enter the cells of the matrix:
matrix[0,0] = 2
matrix[0,1] = 3
matrix[1,0] = 5
matrix[1,1] = 10
matrix[2,0] = 8
matrix[2,1] = 9
 2 3
 5 10
 8 9
```

Maximal Platform in a Matrix – Example

In the next example we will solve another interesting problem: we are given a two-dimensional rectangular array (matrix) of integers and our task is to **find the sub-matrix of size of 2 by 2 with maximum sum** of its elements and to print it to the console.

One solution to the problem might be the following:

MaxPlatform2x2.cs

```
class MaxPlatform2x2
{
  static void Main()
  {
    // Declare and initialize the matrix
    int[,] matrix = {
      { 0, 2, 4, 0, 9, 5 },
      { 7, 1, 3, 3, 2, 1 },
      { 1, 3, 9, 8, 5, 6 },
      { 4, 6, 7, 9, 1, 0 }
    };

    // Find the maximal sum platform of size 2 x 2
    long bestSum = long.MinValue;
    int bestRow = 0;
    int bestCol = 0;

    for (int row = 0; row < matrix.GetLength(0) - 1; row++)
    {
      for (int col = 0; col < matrix.GetLength(1) - 1; col++)
      {
```

```
            long sum = matrix[row, col] + matrix[row, col + 1] +
              matrix[row + 1, col] + matrix[row + 1, col + 1];
            if (sum > bestSum)
            {
              bestSum = sum;
              bestRow = row;
              bestCol = col;
            }
          }
        }

        // Print the result
        Console.WriteLine("The best platform is:");
        Console.WriteLine("  {0} {1}",
          matrix[bestRow, bestCol],
          matrix[bestRow, bestCol + 1]);
        Console.WriteLine("  {0} {1}",
          matrix[bestRow + 1, bestCol],
          matrix[bestRow + 1, bestCol + 1]);
        Console.WriteLine("The maximal sum is: {0}", bestSum);
    }
}
```

If we execute the program, we will see that it works properly:

```
The best platform is:
  9 8
  7 9
The maximal sum is: 33
```

We will explain the algorithm. First we create a two-dimensional array, which contains integer numbers. We declare our auxiliary variables **bestSum**, **bestRow**, **bestCol** and we initialize **bestSum** with the minimal value of type **long** (so any other value is greater than this one). Note that sum of 4 integers may not fit in **int**, so we use **long**.

In the variable **bestSum** we keep the current maximal sum and in **bestRow** and **bestCol** we keep the current best sub-matrix. This means the current row and current column describe the start element for the sub-matrix of size 2 x 2, which is currently found to have the maximal sum of its elements.

To access all elements of a sub-array with a size of 2 by 2 we need the indices of the first element. Having them we can easily access the rest 3 elements:

```
matrix[row, col]
matrix[row, col + 1]
```

```
matrix[row + 1, col]
matrix[row + 1, col + 1]
```

In this example **row** and **col** are the indices of the first element of the sub-matrix with a size of 2 by 2, which is part of the array **matrix**.

After we know how to access all four elements of the matrix with a size of 2 by 2, starting from a particular row and column, we can look at the algorithm, which we will use to find the maximal sub-matrix.

We need to iterate through each 2 x 2 platform in the matrix until we reach the platform with the best sum. We will do this using two nested **for**-loops and two variables **row** and **col**. Note that we are not iterating through the entire matrix, because if we try to access index **row + 1** or **col + 1**, as we are at the last row or column we will go out of the range of the matrix, respectively **System.IndexOutOfRangeException** will be thrown.

We access the neighbor elements of each current element of the sub-matrix and we sum them. Then we check if our current sum is bigger than our current highest sum for the moment. If it is so, our current sum becomes our best sum and our current indices will update **bestRow** and **bestCol**. So, after the entire iteration through the main matrix we will find the maximal sum and the first element of the sub-matrix of size 2 by 2 and its indices.

If there is more than one sub-matrix with the same maximal sum, we will find the one, which appears first.

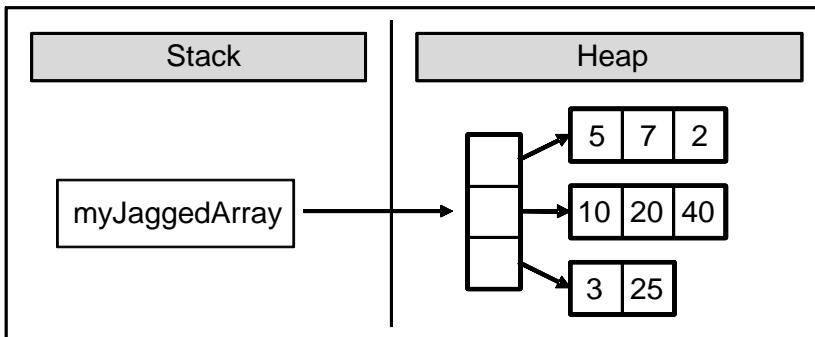
At the end of the example we are printing to the console the requested sub-matrix of size 2 x 2 and its sum of elements in an appropriate way.

Arrays of Arrays

In C# we can have arrays of arrays, which we call **jagged** arrays.

Jagged arrays are **arrays of arrays**, or arrays in which each row contains an array of its own, and that array can have length different than those in the other rows.

Declaration and Allocation an Array of Arrays

The only difference in the declaration of the jagged arrays compared to the regular multidimensional array is that we do not have just one pair of brackets. With the jagged arrays we have a pair brackets per dimension. We **allocate** them this way:

```
int[][] jaggedArray;
jaggedArray = new int[2][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[3];
```

Here is how we declare, allocate and initialize an array of arrays (a jagged array whose elements are arrays of integer values):

```
int[][] myJaggedArray = {
  new int[] {5, 7, 2},
  new int[] {10, 20, 40},
  new int[] {3, 25}
};
```

Memory Allocation

The figure below depicts how the now declared jagged array **myJaggedArray** is allocated in the memory. As we see the jagged arrays are an aggregation of references. A jagged array does not directly contain any arrays, but rather has **elements pointing to them**. The size is unknown and that is why CLR just keeps references to the internal arrays. After we allocate memory for one array-element of the jagged array, then the reference starts pointing to the newly created block in the dynamic memory. The variable **myJaggedArray** is stored in the execution stack of the program and points to a block in the dynamic memory, which contains a sequence of three references to other three blocks in memory; each of them contains an array of integer numbers – the elements of the jagged array:

Initialization and Access to the Elements

We can access elements of the arrays, which are part of the jagged array by using their index. In next example we will access the element with index 3 of the array stored at index 0 in the **myJaggedArray** declared above:

```
myJaggedArray[0][2] = 45;
```

The elements of the jagged array can be **one-dimensional** and **multi-dimensional arrays**. Here is an example for jagged array of two-dimensional arrays:

```
int[][,] jaggedOfMulti = new int[2][,];
```

```
jaggedOfMulti[0] = new int[,] { { 5, 15 }, { 125, 206 } };
jaggedOfMulti[1] = new int[,] { { 3, 4, 5 }, { 7, 8, 9 } };
```

Pascal's Triangle – Example

In the next example we will use a jagged array to generate and visualize the **Pascal's triangle**. As we know from mathematics, the first row of the triangle contains the number 1 and each next number is generated by sum of the two numbers on the row above it. The Pascal's triangle looks like this:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
      . . .
```

To have a Pascal's triangle with a given height, for example 12, we allocate a jagged array **triangle[][]**, which contains 1 element on the zero row, 2 – on first, 3 – on second and so on. First we initialize **triangle[0][0]** = **1** and the rest of the cells will have a default value than **0** by allocation. Then we loop through the rows and from **row** we will get the values for **row+1**. It works with nested **for** loop through the columns on the current row and the following Pascal definitions for values in the triangle: we add the value of the current cell of the current row (**triangle[row][col]**) to the cell below (**triangle[row+1][col]**) and to the cell below on the right (**triangle [row+1][col+1]**). We print using an appropriate number of spaces (using method **PadLeft ()** of class **String**), because we want the result to be aligned.

Here is the code of the described algorithm:

PascalTriangle.cs

```
class PascalTriangle
{
  static void Main()
  {
    const int HEIGHT = 12;

    // Allocate the array in a triangle form
    long[][] triangle = new long[HEIGHT + 1][];

    for (int row = 0; row < HEIGHT; row++)
    {
```

```
      triangle[row] = new long[row + 1];
    }

    // Calculate the Pascal's triangle
    triangle[0][0] = 1;
    for (int row = 0; row < HEIGHT - 1; row++)
    {
      for (int col = 0; col <= row; col++)
      {
        triangle[row + 1][col] += triangle[row][col];
        triangle[row + 1][col + 1] += triangle[row][col];
      }
    }

    // Print the Pascal's triangle
    for (int row = 0; row < HEIGHT; row++)
    {
      Console.Write("".PadLeft((HEIGHT - row) * 2));
      for (int col = 0; col <= row; col++)
      {
        Console.Write("{0,3} ", triangle[row][col]);
      }
      Console.WriteLine();
    }
  }
}
```

If we execute the program, we will see that it is working properly and it generates a **Pascal's triangle** by a given numbers of rows (in our case the **HEIGHT** is 12):

```
                       1
                     1   1
                   1   2   1
                 1   3   3   1
               1   4   6   4   1
             1   5  10  10   5   1
           1   6  15  20  15   6   1
         1   7  21  35  35  21   7   1
       1   8  28  56  70  56  28   8   1
     1   9  36  84 126 126  84  36   9   1
   1  10  45 120 210 252 210 120  45  10   1
 1  11  55 165 330 462 462 330 165  55  11   1
```

Exercises

1. Write a program, which creates an array of **20 elements of type integer** and initializes each of the elements with a value equals to the index of the element multiplied by 5. Print the elements to the console.
2. Write a program, which **reads two arrays** from the console and **checks whether they are equal** (two arrays are equal when they are of equal length and all of their elements, which have the same index, are equal).
3. Write a program, which **compares two arrays of type `char` lexicographically** (character by character) and checks, which one is first in the lexicographical order.
4. Write a program, which finds the **maximal sequence of consecutive equal elements** in an array. E.g.: {1, 1, 2, 3, **2, 2, 2**, 1} → {2, 2, 2}.
5. Write a program, which finds the **maximal sequence** of consecutively placed **increasing** integers. Example: {3, **2, 3, 4**, 2, 2, 4} → {2, 3, 4}.
6. Write a program, which finds the **maximal sequence of increasing elements** in an array **`arr[n]`**. It is not necessary the elements to be consecutively placed. E.g.: {9, 6, **2**, 7, **4**, 7, **6**, 5, **8**, 4} → {2, 4, 6, 8}.
7. Write a program, which reads from the console two integer numbers **N** and **K** (K<N) and array of N integers. Find those **K consecutive elements** in the array, which have **maximal sum**.
8. **Sorting an array** means to arrange its elements in an increasing (or decreasing) order. Write a program, which sorts an array using the algorithm "**`selection sort`**".
9. Write a program, which finds a **subsequence of numbers with maximal sum**. E.g.: {2, 3, -6, -1, **2, -1, 6, 4**, -8, 8} → **11**
10. Write a program, which finds the **most frequently occurring** element in an array. Example: {**4**, 1, 1, **4**, 2, 3, **4**, **4**, 1, 2, **4**, 9, 3} → 4 (5 times).
11. Write a program to find a sequence of neighbor numbers in an array, which has a **sum of certain number S**. Example: {4, 3, 1, **4, 2, 5**, 8}, S=**11** → {4, 2, 5}.
12. Write a program, which creates **square matrices** like those in the **figures below** and prints them formatted to the console. The size of the matrices will be read from the console. See the examples for matrices with size of 4 x 4 and make the other sizes in a similar fashion:

a)

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

b)

1	8	9	16
2	7	10	15
3	6	11	14
4	5	12	13

c)

7	11	14	16
4	8	12	15
2	5	9	13
1	3	6	10

d)*

1	12	11	10
2	13	16	9
3	14	15	8
4	5	6	7

13. Write a program, which creates a rectangular array with size of **n** by **m** elements. The dimensions and the elements should be read from the console. Find a **platform with size of (3, 3) with a maximal sum**.

14. Write a program, which finds the **longest sequence of equal string** elements in a matrix. A sequence in a matrix we define as a set of neighbor elements **on the same row, column or diagonal**.

ha	fifi	ho	hi
fo	ha	hi	xx
xxx	ho	ha	xx

→ ha, ha, ha

s	qq	s
pp	pp	s
pp	qq	s

→ s, s, s

15. Write a program, which creates an array containing **all Latin letters**. The user inputs **a word** from the console and as result the program prints to the console the **indices of the letters from the word**.

16. Write a program, which uses a **binary search** in a **sorted** array of integer numbers to find a certain element.

17. Write a program, which sorts an array of integer elements using a **"merge sort"** algorithm.

18. Write a program, which sorts an array of integer elements using a "**quick sort"** algorithm.

19. Write a program, which finds **all prime numbers** in the range [1…10,000,000].

20. * Write a program, which checks whether there is a **subset** of given array of **N** elements, which has a **sum S**. The numbers **N**, **S** and the array values are read from the console. Same number can be used many times.

    Example: {2, **1, 2**, 4, 3, **5**, 2, **6**}, **S** = **14** → yes (1 + 2 + 5 + 6 = 14)

21. Write a program which by given **N** numbers, **K** and **S**, finds **K** elements out of the **N** numbers, the sum of which is exactly **S** or says it is not possible.

    Example: {3, **1,** 2, **4**, **9**, 6}, **S** = **14**, **K** = **3** → yes (1 + 2 + 4 = 14)

22. Write a program, which reads an array of integer numbers from the console and **removes a minimal number of elements** in such a way that **the remaining array is sorted** in an increasing order.

    Example: {6, **1,** 4, **3**, 0, **3**, 6, **4**, **5**} → {1, 3, 3, 4, 5}

23. Write a program, which reads the integer numbers **N** and **K** from the console and prints **all variations of K elements of the numbers in the interval [1…N]**. Example: N = 3, K = 2 → {1, 1}, {1, 2}, {1, 3}, {2, 1}, {2, 2}, {2, 3}, {3, 1}, {3, 2}, {3, 3}.

24. Write a program, which reads an integer number **N** from the console and prints **all combinations of K elements of numbers in range [1 … N]**. Example: N = 5, K = 2 → {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}.

25. *Write a program, which finds in a given matrix the **largest area of equal numbers**. We define an **area** in the matrix as a set of neighbor cells (by row and column). Here is one example with an area containing 13 elements with equal value of 3:

1	3	2	2	2	4
3	3	3	2	4	4
4	3	1	2	3	3
4	3	1	3	3	1
4	3	3	3	1	1

→ 13

Solutions and Guidelines

1. Use an `int[]` array and a `for`-loop.

2. Two arrays are **equal** if they have the same value for the **length** and the **values for their elements**. You can check for the second condition using a `for`-loop.

3. In **lexicographic order** the elements are **compared one by one** starting from the very left. If the elements are not the same, the array, whose element is smaller (comes earlier in the alphabet), comes first. If the elements are equal, the next character is compared. If the end of one of the arrays is reached, without finding different elements, the shorter array is the **smaller** (comes earlier lexicographically). If all elements are equal, the arrays are **equal**.

4. **Scan the array from left to right**. Every time when the current number is different from the one before it, a **new sequence starts**. If the current element is equal to the one before it, it is a **continuation of the same sequence**. So, if we keep the index of the **start position** of the current sequence (in the beginning it is 0) in **start** and the **length of the current sequence** (in the beginning it is 1) in **len**, we can find all

sequences of equal elements and their lengths. We can easily keep the shortest one in two additional variables – **bestStart** and **bestLen**.

5. This exercise is **very similar to the previous one**, but we have a continuation of the current sequence when **the next element is bigger**.

6. We can solve the problem with **two nested loops** and one more array **len[0…n-1]**. In the array **len[i]** we can keep the length of the longest consecutively increasing sequence, which starts somewhere in the array (it does not matter where exactly) and ends with the element **arr[i]**. Therefore **len[0]=1**, **len[x]** is the maximal sum **max(1 + len[prev])**, where **prev < x** and **arr[prev] < arr[x]**. Following the definition, we can calculate **len[0…n-1]** with two nested loops: the **outer loop** will iterate through the array **from left to right** with the loop variable **x**. The **inner loop** will iterate through the array from the start to position **x-1** and searches for the element **prev** with maximal value of **len[prev]**, where **arr[prev] < arr[x]**. After the search, we initialize **len[x]** with 1 + the biggest found value of **len[prev]** or with **1**, if such a value is not found.

   The described algorithm **finds the lengths of all maximal ascending sequences**, which end at each of the elements. The biggest one of these values is the length of the **longest increasing sequence**. If we need to find **the elements themselves**, which compose that longest sequence, we can start from the element, where the sequence ends (at index **x**), we can print it and we can search for a previous element (**prev**). By definition **prev < x** and **len[x] = 1 + len[prev]** so we can find **prev** with a **for**-loop from **1** to **x-1**. After that we can repeat the same for **x=prev**. By finding and printing the previous element (**prev**) many times until it exists, we can find **the elements, which compose the longest sequence** in reversed order (from the last to the first).

7. You can find out which sequence of the sequences of **K** numbers has the biggest sum by **checking the sums of all of those sequences**. The first sequence starts at index **0** and finishes at index **K-1** and has sum **S**. Then the second one starts at index **1** and ends at index **K** and we can find its sum using **S** by subtracting the element at index **0** and adding the element at index **K**. In this way we can reach the end of the sequence.

8. **Find in Internet information about "Selection sort"** and its C# implementations. Briefly the idea is to find the smallest element and to place it at position 0 (through **swapping**) then to find the smallest number excluding the first and place it at position 1 and so on, until the entire array is arranged in ascending order.

9. There are **two ways** to solve this problem. The first way is to use **brute force method**, which in this case means that using **two nested loops** we check every possible start and end and its corresponding sum.

   The second way is to **use one loop through the array** to scan it from left to right and sum the elements. Once we get a **negative sum**, we can

restart summing from the next element. Think why this is correct! At each step we check if the current sum is greater than the current max.

10. This exercise **can be solved in a couple of ways**. One of them is the following: get the first number and check how many times it is repeated in the array and store this number in a variable. After a repeated number is found we change its value to **int.MinValue**. Then pass to the **next number** and do the same with it. The current number is remembered if its occurrences are maximal. As you may guess, when a number equal to **int.MinValue** is found (already processed number) we should skip it.

    Another solution is to **sort the numbers** in ascending order and then the elements with same value will be placed next to each other. So, basically we then **find the longest sequence of neighbor equal elements**.

11. This exercise can be solved with **two nested loops**. The **first loop** assigns a starting index. The **second loop** sums the elements from the starting index to the right until this partial sum reaches or is greater than **S**. If the sum is equal to **S**, we will remember the starting index (from the first loop) and the ending index (from the second loop).

    If all numbers are positive, there is a **much faster algorithm**. We **sum all numbers from left to the right**, starting from zero. If the current sum becomes greater than **S** during the summation, we remove the leftmost number in the sequence and we subtract it from the sum. If the current sum is still greater than **S**, we remove the next leftmost number and do that until the current sum becomes smaller than **S**. When the sum becomes smaller than **S** we **add the next number on right**. If we find a sum equal to **S**, we print the sum and the sequence to the console. So this solution uses **just with one scan** through the elements in the array.

12. a), b), c) Think about appropriate **ways for iterating through the matrices** with **two nested loops**.

    d) We can start from (0, 0) and go **down N times**. Therefore, go to the **right N-1 times**, after that **up N-1 times**, after that **left N-2 times**, after that **down N-2** times and etc. At each iteration we place the next number in a sequence 1, 2, 3, …, N in the cell, which we are leaving.

13. Modify the example about **maximal platform with size of 2 by 2**.

14. Check every element in a diagonal line, a row and a column until you get a **sequence**. If you get a sequence, check whether this sequence is longer than the currently longest sequence.

15. We can solve this problem with **two nested for-loops** (one for the words and one for the letters of the current word). There is a solution without using an array: we can calculate the index of a given uppercase Latin letter **ch** using the expression: **(int) ch – (int) 'A'**.

16. Find on the Internet information about **the algorithm "binary search"**. Note that binary search works only on **sorted arrays**.

17. Find on the Internet information about **the algorithm "merge sort"** and its implementations in C#. It is a bit **complicated to write merge sort efficiently**. You can have **3 preallocated arrays** when merging arrays: **two arrays for keeping the numbers for merging** and a **result array**. Thus you will never allocate new arrays during the algorithm's execution. The arrays will be allocated just once at the start and you will just change their purpose (**swap them**) during the algorithm execution.

18. Find information about **the "quick sort" algorithm** in Internet and its C# implementations. It can be best implemented by using **recursion**. See the chapter "Recursion" to read about recursive algorithms. Generally at each step you choose an element called **pivot** and reorder the array into two sections: at the **left side** move all **elements ≤ pivot** and at the **right side** move all **elements > pivot**. Finally **run the quicksort algorithm recursively** over the left and the right sides.

19. Find on the Internet information about "**The sieve of Erathostenes**" (you have probably heard about it in math classes in high-school).

20. **Generate all possible sums** this way: take all the numbers and mark them as "**possible sum**". Then take every number k_0, k_2, …, k_{n-1} and for each already marked "possible sum" **p**, mark as possible the sum $p+k_i$. If at some step you get **S**, a solution is found. You can keep track of the "possible sums" either in a **bool[]** array **possible[]**, where each index is a possible sum, or in a more complex data structure like **Set<int>**. Once you have **possible[S] == true**, you can find a number k_i such that **possible[S-**k_i**] == true**, print k_i and subtract it from **S**. Repeat the same to find the next k_i and print and subtract is again, until **S** reaches **0**.

    Another algorithm: generate **all possible subsets** of the numbers by a **for**-loop from 0 to 2^N-1. If we have a number **p**, take its binary representation (which consists of **exactly N bits**) and sum the numbers that correspond to 1 in the binary representation of **p** (with a **nested loop** from 0 to N-1). Thus all possible sums will be generated and if some of them is **S**, it can be printed. Note that **this algorithm is slow** (needs exponential time and cannot run for 100 or 1000 elements). It also does not allow using the same array element twice in the sum.

21. **See the previous problem**. Generate all subsets **of exactly K elements** (the **second algorithm**) and check if their sum is equal to **S**.

    Try in the **first algorithm** to think how to keep the count of the numbers used in the sum in order to take exactly **K** numbers. Can you define a matrix **possible[p, n]** to keep whether the number **p** can be obtained as a sum of the first **n** numbers (the numbers k_0, k_2, …, k_{n-1})?

22. Use **dynamic programming** to find the **longest increasing subsequence** in the input sequence **arr[]**, just like in problem #6. The elements not included in the maximal increasing sequence should be removed in order the array to become sorted.

23. Start from the **first variation** in the lexicographical order: **{1, 1, …} K** times. Think of this as **k-digit number**. To **obtain the next variation**, **increase the last digit**. If it becomes greater than N, change it to 1 and increase the next digit on the left. Do the same on the left until the first digit goes greater than N.

24. Modify the algorithm from **the previous problem** in the following way: start from {1, 2, …, N} and increase the last digit (with the digits at the left when required), but always keep all elements in the array in **ascending order** (element **p[i]** should start increasing from **p[i-1]+1**).

25. This is a little bit more difficult. You can use different **graph traversal algorithms** like "**DFS**" (**Depth-First-Search**) and "**BFS**" (**Breadth-First-Search**) to go through all the cells in certain area starting from any cell that belongs to it. If you have an **area traversal algorithm** (like DFS), run it several times starting from unvisited cell and mark the cells of the traversed area as **visited**. Repeat this **until all cells become visited**. Read later in this book about DFS and BFS in the chapter "Trees and Graphs" or find information about these algorithms in Internet.

Chapter 8. Numeral Systems

In This Chapter

In this chapter we will take a look at **working with different numeral systems** and **how numbers are represented** in them. We will pay more attention to how numbers are represented in **decimal**, **binary** and **hexadecimal** numeral systems, since they are most widely used in computers and programming. We will also explain the different ways for encoding numeral data in computers – signed or unsigned integers and the different types of real numbers.

History in a Nutshell

Different numeral systems have been used since the **ancient times**. This claim is supported by the fact that in ancient Egypt people used sun dials, which measure time with the help of numeral systems. Most historians believe that ancient Egyptians are the first civilization, which divided the day into smaller parts. They accomplished this by using the first sun dials, which were nothing more than a simple pole stuck in the ground, oriented by the length and direction of the shadow.

Later a better **sundial** was invented, which looked like the letter T and divided the time between sunrise and sunset into 12 parts. This proves the use of the duodecimal system in ancient Egypt, the importance of the number 12 is usually related to the fact that moon cycles in a single year are 12 or the number of phalanxes found in the fingers of one hand (four in each finger, excluding the thumb).

In modern times, the **decimal system** is the most widely spread numeral system. Maybe this is due to the fact that it enables people to count by using the fingers on their hands.

Ancient civilizations divided the day into smaller parts by using different numeral systems – **duodecimal** and **sexagesimal** with bases **12** and **60** respectively. Greek astronomers such as Hipparchus used astronomical approaches, which were earlier used by the Babylonians in Mesopotamia. The Babylonians did astronomical calculations using the sexagesimal system, which they had inherited from the Sumerians, who had developed it on their own around 2000 B.C. It is not known exactly why the number 60 was chosen for a base of the numeral system but it is important to note that this system is very appropriate for the representation of fractions, because the number 60 is the smallest number that can be divided by 1, 2, 3, 4, 5, 6, 10, 12, 15, 20 and 30 without a remainder.

Applications of the Sexagesimal Numeral System

The **sexagesimal system** is still used today for measuring angles, geographical coordinates and time. It still finds application on the watch dial and the sphere of the geographical globe. The sexagesimal system was used by Eratosthenes for dividing a circumference into 60 parts in order to create an early system of geographical latitudes, made up from horizontal lines passing through places well known in the past.

One century after Eratosthenes, Hipparchus standardized these lines by making them parallel and conformable to the geometry of the Earth. He introduced a **system of geographical longitude lines**, which included 360 degrees and respectively passed from north to south and pole to pole. In the book "Almagest" (150 A.D.), Claudius Ptolemy further developed Hipparchus' studies by dividing the 360 degrees of geographical latitude and longitude into other smaller parts. He divided each of the degrees into 60 equal parts, each of which was later divided again into 60 smaller and equal parts. The parts created by the division were called partes minutiae primae, or "first minute" and respectively partes minutiae secundae, or "second minute". These parts are still used today and are called "minutes" and "seconds" respectively.

Short Summary

We took a short historical trip through the millennia, which helped us learn that numeral systems were created, used and developed as far back as the Sumerians. The presented facts explain why a **day contains (only) 24 hours**, the **hour has 60 minutes** and the **minute has 60 seconds**. This is a result of the fact that the ancient Egyptians divided the day after they had started using the duodecimal numeral system. The division of hours and minutes into 60 equal parts is a result of the work of ancient Greek astronomers, who did their calculations using the sexagesimal numeral system, which was created by the Sumerians and used by the Babylonians.

Numeral Systems

So far we have taken a look at the history of numeral systems. Let's now take a detailed look at what they really are and what is **their role in computing**.

What Are Numeral Systems?

Numeral systems are a way of representing numbers by a finite type-set of graphical signs called digits. We must add to them the rules for depicting numbers. The characters, which are used to depict numbers in a given numeral system, can be perceived as that system's **alphabet**.

During the different stages of the development of human civilization, various numeral systems had gained popularity. We must note that today the most widely spread one is the **Arabic numeral system**. It uses the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, as its alphabet. (An interesting fact is that the depiction of Arabic numerals in modern times is different from the ten digits mentioned

above but in spite of all they are still referred to the same numeral system – the decimal one).

Beside an alphabet, every numeral system has a **base**. The base is a number equal to the different digits used by the system for depicting the numbers in it. For example, the Arabic numeral system is **decimal** because it has **10 digits**. A random number can be chosen as a base, which has an absolute value different than 1 and 0. It can also be a real or a complex number with a sign.

A practical question we can ask is: **which is the best numeral system** that we should use? To answer it, we must decide what the optimal way to depict a number (the digit count in the number) is and the number of digits the given numeral system uses – its base. Mathematically it can be proven that the best ratio between the length of depiction and the number of used digits is accomplished by using Euler's number (e = 2,718281828), which is the base of natural logarithms.

Working in a system with such base e is **extremely inconvenient and impractical** because that number cannot be represented as a ratio of two natural numbers. This gives us grounds to conclude that the optimal base of a numeral system is either 2 or 3.

Although the number 3 is closer to the Neper number, it is unsuitable for technical implementation. Because of that the **binary numeral system** is the only one suitable for practical use and it is used in the modern computers and electronic devices.

Positional Numeral Systems

A **positional numeral system** is a system, in which the **position of the digits** is significant for the value of the number. This means that the value of the digits in the number is not strictly defined and depends on which position the given digit is. For example, in the number 351 the digit 1 has a value of 1, while in the number 1024 it has a value of 1000. We must note that the bases of the numeral systems are applicable only with positional numeral systems. In a positional numeral system the number $A_{(p)} = (a_{(n)}a_{(n-1)}...a_{(0)},a_{(-1)}a_{(-2)}...a_{(-k)})$ can be represented in the following way:

$$A_{(p)} = \sum_{m=n}^{-k} a_m T_m$$

In this sum T_m has the meaning of a weight factor for the m^{th} digit of the number. In most cases $T_m = P^m$, which means that:

$$A_{(p)} = \sum_{m=n}^{-k} a_m P^m$$

Formed using the sum above, the number $A_{(p)}$ is respectively made up from its whole part ($a_{(n)}a_{(n-1)}...a_{(0)}$) and its fraction ($a_{(-1)}a_{(-2)}...a_{(-k)}$), where every *a* belongs to the multitude of the natural numbers *M={0, 1, 2, …, p-1}*. We can easily see that in positional numeral systems the value of each digit is the-base-of-the-system times bigger than the one before it (the digit to the right, which is the lower-order digit). As a direct result from this we must add one to the left (higher-order) digit, if we need to note a digit in the current digit that is bigger than the base. The systems with bases of 2, 8, 10 and 16 have become wide spread in computing devices. In the table below we can see their notation of **the numbers from 0 to 15**:

Binary	Octal	Decimal	Hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Non-Positional Numeral Systems

Besides the positional numeral systems, there are also **non-positional numeral systems**, in which the value of each digit is a constant and does not strictly depend on its position in the number. Such numeral systems are the **Roman** and **Greek** numeral systems. All non-positional numeral systems have a common drawback – the notation of big numbers in them is very inefficient. As a result of this drawback, they have gained only limited use. This could often lead to inaccuracy when determining the value of numbers. We will take a very brief look at the Roman and Greek numeral systems.

Roman Numeral System

The **Roman** numeral system uses sequences of the following symbols to represent the numbers:

Roman Digit	Decimal Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

As we have already mentioned, in this numeral system the position of the digit has no significance for the value of the number and for determining the value, the following rules are applied:

1. If two consecutively represented Roman digits are in such order that the value of the first one is bigger or equal to the value of the second one, their values are added. Example:

   The number III=3, but the number MMD=2500.

2. If two consecutively represented roman digits are in increasing order of their values, they are subtracted. Example:

   The number IX=9, the number MXL=1040, but the number MXXIV=1024.

Greek Numeral System

The **Greek** numeral system is a decimal system, in which a grouping of fives is done. It uses the following digits:

Greek Digit	Decimal Value
I	1
Γ	5
Δ	10
Η	100
Χ	1,000
Μ	10,000

As we can see in the table, one is represented with a vertical line, five with the letter Γ, and the powers of 10 with the first letter of the corresponding Greek word.

Here are some examples of numbers in this system:

- ΓΔ = 50 = 5 x 10
- ΓΗ = 500 = 5 x 100
- ΓΧ = 5000 = 5 x 1,000
- ΓΜ = 50,000 = 5 x 10,000

The Binary Numeral System – Foundation of Computing Technology

The binary numeral system is the system, which is used to represent and process numbers in modern computing machines. The main reason it is so widely spread is explained with the fact that devices with two stable states are very simple to implement and the production costs of binary arithmetic devices are very low.

The binary digits **0** and **1** can be easily represented in the computing machines as "current" and "no current", or as "+5V" and "-5V".

Along with its advantages, the binary system for number notation in computers has its drawbacks, too. One of its biggest practical flaws is that numbers represented in binary numeral system are very long, meaning they have a large number of bits. This makes it inconvenient for direct use by humans. To avoid this disadvantage, systems with larger bases are used in practice.

Decimal Numbers

Numbers represented in the **decimal numeral system**, are given in a primal appearance, meaning that they are easy to be understood by humans. This numeral system has the number 10 for a base. The numbers represented in it are ordered by the powers of the number 10. The lowest-order digit (first from right to left) of the decimal numbers is used to represent the ones ($10^0=1$), the next one to represent the tens ($10^1=10$), the next one to represent the hundreds ($10^2=100$), and so on. In other words – every following digit is ten times bigger than the one preceding it. The sum of the separate digits determines the value of the number. We will take the number 95031 as an example, which can be represented in the decimal numeral system as:

$$95031 = (\mathbf{9}\times10^4) + (\mathbf{5}\times10^3) + (\mathbf{0}\times10^2) + (\mathbf{3}\times10^1) + (\mathbf{1}\times10^0)$$

Represented that way, the number 95031 is presented in a natural way for humans because the principles of the decimal numeral system have been accepted as fundamental for people.

	The discussed approaches are valid for the other numeral systems, too. They have the same logical setting but are applied to a system with a different base. The last statement

	is true for the binary and hexadecimal numeral systems, which we will discuss in details in a little bit.

Binary Numbers

The numbers represented in the **binary numeral system** are represented in a secondary aspect – which means that they are easy to be understood by the computing machine. They are a bit harder to be understood by people. To represent a binary number, the binary numeral system is used, which has the number 2 for a base. The numbers represented in it are ordered by the powers of two. Only the digits 0 and 1 are used for their notation.

Usually, when a number is represented in a numeral system other than decimal, the numeral system's base is added as an index in brackets next to the number. For example, with this notation $\mathbf{1110_{(2)}}$ we indicate a number in the binary numeral system. If no numeral system is explicitly specified, it is accepted that the number is in the decimal system. The number is pronounced by reading its digits in sequence from left to right (we read from the highest-order to the lowest-order bit).

Like with decimal numbers, each binary number being looked at from right to left is represented by **a power of the number 2** in the respected sequence. The lowest-order position in a binary number corresponds to the zero power ($2^0=1$), the second position corresponds to 2 to the first power ($2^1=2$), the third position corresponds to 2 to the second power ($2^2=4$), and so on. If the number is 8 bits long, the last bit is 2 to the seventh power ($2^7=128$). If the number has 16 bits, the last bit is 2 to the fifteenth power. By using 8 binary digits (0 or 1) we can represent a total of 256 numbers, because $2^8=256$. By using 16 binary digits we can represent a total of 65536 numbers, because $2^{16}=65536$.

Let's look at some examples of numbers in the binary numeral system. Take, for example, the decimal number **148**. It is composed of three digits: **1**, **4** and **8**, and it corresponds to the following binary number:

$$\mathbf{10010100_{(2)}}$$

$$\mathbf{148} = (\mathbf{1}\times 2^7) + (\mathbf{1}\times 2^4) + (\mathbf{1}\times 2^2)$$

The full notation of the number is depicted in the following table:

Number	**1**	**0**	**0**	**1**	**0**	**1**	**0**	**0**
Power	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value	1×2^7 $= 128$	0×2^6 $= 0$	0×2^5 $= 0$	1×2^4 $= 16$	0×2^3 $= 0$	1×2^2 $= 4$	0×2^1 $= 0$	0×2^0 $= 0$

The sequence of eight zeros or ones represents one **byte**, an ordinary eight bit binary number. All numbers from 0 to 255 including can be represented in a single byte. In most cases this is not enough; as a result several consecutive bytes can be used to represent a big number. Two bytes form the

so called "machine word" (**word**), which corresponds to 16 bits (in 16-bit computing machines). Besides it, computing machines use the so called **double word** or **dword**, corresponding to 32 bits.

	If a binary number ends in 0 it is even, if it ends in 1 it is odd.

Converting From Binary to Decimal Numeral System

When turning from binary to decimal numeral system, we do a **conversion of a binary number to a decimal number**. Every number can be converted from one numeral system to another by doing a sequence of operations that are possible in both numeral systems. As we have already mentioned, numbers in the binary system consist of binary digits, which are ordered by the powers of 2. Let's take the number $\mathbf{11001}_{(2)}$. Converting into decimal is done by calculating the following sum:

$$\mathbf{11001}_{(2)} = \mathbf{1}\times2^4 + \mathbf{1}\times2^3 + \mathbf{0}\times2^2 + \mathbf{0}\times2^1 + \mathbf{1}\times2^0 =$$

$$= \mathbf{16}_{(10)} + \mathbf{8}_{(10)} + \mathbf{1}_{(10)} = \mathbf{25}_{(10)}$$

From this follows that $\mathbf{11001}_{(2)} = \mathbf{25}_{(10)}$

In other words – every single binary digit is **multiplied by 2 raised to the power of the position it is in**. In the end all of the numbers resulting from the binary digits are added up to get the decimal value of the binary number.

Horner Scheme

Another method of conversion exists, known as the **Horner Scheme**. When using it, we multiply the left most digit by 2 and add it to the one to its right. We multiply this result by two and the neighboring digit (one to the right) is added. This is repeated until all the digits in the number have been exhausted and we add the last digit without multiplying it. Here is an example:

$\mathbf{1001}_{(2)} = ((1 \times 2 + 0) \times 2 + 0) \times 2 + 1 = 2 \times 2 \times 2 + 1 = \mathbf{9}$

Converting from Decimal to Binary Numeral System

When transitioning **from decimal to binary numeral system**, we convert a decimal number into a binary one. To accomplish this, we divide it by 2 with a remainder. This is how we get the quotient and the remainder, which is separated.

Let's use the number 148 again as an example. We do an integer division by the base we want to convert to (in this case it is 2). After that using the remainders of the division (they will always be either zero or one), we represent the converted number. We continue dividing until we get a zero quotient. Here is an example:

148:2=74 with remainder 0;

74:2=37 with remainder 0;

37:2=18 with remainder 1;

18:2=9 with remainder 0;

9:2=4 with remainder 1;

4:2=2 with remainder 0;

2:2=1 with remainder 0;

1:2=0 with remainder 1;

After we are done with the division, we represent the remainders in reverse order as follows:

10010100

i.e. $148_{(10)} = 10010100_{(2)}$

Operations with Binary Numbers

The arithmetical rules of addition, subtraction and multiplication are valid for a single digit of binary numbers:

0 + 0 = 0	0 - 0 = 0	0 × 0 = 0
1 + 0 = 1	1 - 0 = 1	1 × 0 = 0
0 + 1 = 1	1 - 1 = 0	0 × 1 = 0
1 + 1 = 10	10 - 1 = 1	1 × 1 = 1

In addition, with binary numbers we can also do logical operations such as logical multiplication (conjunction), logical addition (disjunction) and the sum of modulo two (exclusive or).

We must also note that when we are doing arithmetic operations with multi-order numbers we must take into account the connection between the separate orders by transfer or loan, when doing addition or subtraction respectively. Let’s take a look at some details regarding bitwise operators.

Bitwise "and"

The bitwise **AND** operator can be used for checking the value of a given bit in a number. For example, if we want to check if a given number is even (we check if the lowest-order bit is **1**):

1011101**1** **AND** 0000000**1** = 00000001

The result is 1, which means that the number is odd (if the result was 0 the number would be even).

In C# the bitwise "and" is represented with **&** and is used like this:

```
int result = integer1 & integer2;
```

Bitwise "or"

The bitwise **OR** operator can be used if we want, for example, to "raise" a given bit to 1:

10111**0**11 **OR** 00000**1**00 = 10111111

Bitwise "or" in C# is represented with **|** and is used like this:

```
int result = integer1 | integer2;
```

Bitwise "exclusive or"

The bitwise operator **XOR** – every binary digit is processed separately, and when we have a 0 in the second operand, the corresponding value of the bit in the first operand is copied in the result. At every position that has a value of 1 in the second operand, we reverse the value of the corresponding position in the first operand and represent it in the result:

10111011 **XOR** 01010101 = 11101110

In C# the notation of the "exclusive or" operator is **^**:

```
int result = integer1 ^ integer2;
```

Bitwise Negation

The bitwise operator **NOT** – this is a unary operator, which means that it is applied to a single operand. What it does is to reverse every bit of the given binary number to its opposite value:

NOT 10111011 = 01000100

In C# the bitwise negation is represented with **~**:

```
int result = ~integer1;
```

Hexadecimal Numbers

With **hexadecimal numbers** we have the **number 16** for a system base, which implies the use of 16 digits to represent all possible values from 0 to 15 inclusive. As we have already shown in one of the tables in the previous sections, for notating numbers in the hexadecimal system, we use the digits from 0 to 9 and the Latin numbers from A to F. Each of them has the corresponding value:

A=10, B=11, C=12, D=13, E=14, F=15

We can give the following example for hexadecimal numbers: D2, 1F2F1, D1E and so on.

Transition to decimal system is done by multiplying the value of the right most digit by 16^0, the next one to the left by 16^1, the next one to the left by 16^2 and so on, and adding them all up in the end. Example:

$D1E_{(16)} = E*16^0 + 1*16^1 + D*16^2 = 14*1 + 1*16 + 13*256 = 3358_{(10)}$.

Transition from decimal to hexadecimal numeral system is done by dividing the decimal number by 16 and taking the remainders in reverse order. Example:

3358 / 16 = 209 + remainder 14 (E)

209 / 16 = 13 + remainder 1 (1)

13 / 16 = 0 + remainder 13 (D)

We take the remainders in reverse order and get the number $D1E_{(16)}$.

Fast Transition from Binary to Hexadecimal Numbers

The fast conversion **from binary to hexadecimal numbers** can be quickly and easily done by dividing the binary number into groups of four bits (splitting it into half-bytes). If the number of digits is not divisible by four, leading zeros in the highest-orders are added. After the division and the eventual addition of zeros, all the groups are replaced with their corresponding digits. Here is an example:

Let's look at the following: $1110011110_{(2)}$.

1. We divide it into half-bytes and add the leading zeros

   Example: 0011 1001 1110.

2. We replace every half-byte with the corresponding hexadecimal digit and we get $39E_{(16)}$.

Therefore $1110011110_{(2)} = 39E_{(16)}$.

Numeral Systems – Summary

As a summary, we will formulate again in a short but clear manner the algorithms used for transitioning from one positional numeral system to another:

- Transitioning **from a decimal to a k-based numeral system** is done by consecutively dividing the decimal to the base of the k system and the remainders (their corresponding digit in the k based system) are accumulated in reverse order.

- Transitioning **from a k-based numeral system to decimal** is done by multiplying the last digit of the k-based number by k^0, the one before it by k^1, the next one by k^2 and so on, and the products are the added up.

- Transitioning **from a k-based numeral system to a p-based numeral system** is done by intermediately converting to the decimal system (excluding hexadecimal and binary numeral systems).
- Transitioning **from a binary to hexadecimal numeral system and back** is done by converting each sequence of 4 binary bits into its corresponding hexadecimal number and vice versa.

Representation of Numbers

Binary code is used to store data in the operating memory of computing machines. Depending on the type of data we want to store (strings, integers or real numbers with an integral and fractal part) information is represented in a particular manner. It is determined by the data type.

Even a programmer using a high level language must know how the data is allocated in the operating memory of the machine. This is also relevant to the cases when the data is stored on an external carrier, because when it is processed, it will be situated in the operating memory.

In the current section we will take a look at the **different ways to present and process different types of data**. In general they are based on the concepts of bit, byte and machine word.

Bit is a binary unit of information with a value of either 0 or 1.

Information in the memory is grouped in sequences of 8 bits, which form a single **byte**.

For an arithmetic device to process the data, it must be presented in the memory by a set number of bytes (2, 4 or 8), which form a machine word. These are concepts, which every programmer must know and understand.

Representing Integer Numbers in the Memory

One of the things we have not discussed so far is the sign of numbers. Integers can be represented in the memory in two ways: **with a sign** or **without a sign**. When numbers are represented with a sign, a signed order is introduced. It is the highest-order and has the value of 1 for negative numbers and the value of 0 for positive numbers. The rest of the orders are informational and only represent (contain) the value of the number. In the case of a number without a sign, all bits are used to represent its value.

Unsigned Integers

For **unsigned integers** 1, 2, 4 or 8 bytes are allocated in the memory. Depending on the number of bytes used in the notation of a given number, different scopes of representation with variable size are formed. Through *n* bytes all integers in the range [0, 2^n-1] can be represented. The following table shows the range of the values of unsigned integers:

Number of bytes for representing the number in the memory	Range	
	Notation with order	**Regular notation**
1	$0 \div 2^{8}-1$	0 ÷ 255
2	$0 \div 2^{16}-1$	0 ÷ 65,535
4	$0 \div 2^{32}-1$	0 ÷ 4,294,967,295
8	$0 \div 2^{64}-1$	0 ÷ 18,446,744,073,709,551,615

We will give as an example a single-byte and a double-byte representation of the number 158, whose binary notation is the following $10011110_{(2)}$:

1. Representation with 1 byte:

1	0	0	1	1	1	1	0

2. Representation with 2 bytes:

0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0

Representing Negative Numbers

For negative numbers 1, 2, 4 or 8 bytes are allocated in the memory of the computer, while the highest-order (**the left most bit**) has a signature meaning and carries the information about the sign of the number. As we have already mentioned, when the signature bit has a value of 1, the number is negative, otherwise it is positive.

The next table shows the range of the values of the signed integer numbers in the computer according to the number of bytes used for their notation:

Number of bytes for representing the number in the memory	Rank	
	Notation with order	**Regular notation**
1	$-2^{7} \div 2^{7}-1$	-128 ÷ 127
2	$-2^{15} \div 2^{15}-1$	-32,768 ÷ 32,767
4	$-2^{31} \div 2^{31}-1$	-2,147,483,648 ÷ 2,147,483,647
8	$-2^{63} \div 2^{63}-1$	-9,223,372,036,854,775,808 ÷ 9,223,372,036,854,775,807

To encode negative numbers, **straight**, **reversed** and **additional code** is used. In all these three notations signed integers are within the range: [-2^{n-1},

2^{n-1}-1]. Positive numbers are always represented in the same way and the straight, reversed and additional code all coincide for them.

Straight code (**signed magnitude**) is the simplest representation of the number. The highest-order bit carries the sign and the rest of the bits hold the absolute value of the number. Here are some examples:

The number 3 in signed magnitude is represented as an eight-bit-long number 00000011.

The number -3 in signed magnitude is represented in an eight-bit-long number as 10000011.

Reversed code (**one's complement**) is formed from the signed magnitude of the number by inversion (replacing all ones with zeros and vice-versa). This code is not convenient for the arithmetical operations addition and subtraction because it is executed in a different way if subtraction is necessary. Moreover the sign carrying bits need to be processed separately from the information carrying ones. This drawback is avoided by using additional code, which instead of subtraction implements addition with a negative number. The latter is depicted by its addition, i.e. the difference between 2^n and the number itself. Example:

The number -127 in signed magnitude is represented as 1 1111111 and in one's complement as 1 0000000.

The number 3 in signed magnitude is represented as 0 0000011, and in one's complement looks like 0 1111100.

Additional code (**two's complement**) is a number in reversed code to which one is added (through addition). Example:

The number -127 is represented with additional code as 1 0000001.

In the **Binary Coded Decimal**, also known as **BCD** code, in one byte two decimal digits are recorded. This is achieved by encoding a single decimal digit in each half-byte. Numbers presented in this way can be packed, which means that they can be represented in a packed format. If we represent a single decimal digit in one byte we get a non-packed format.

Modern microprocessors use one or several of the discussed codes to present negative numbers, the most widespread method is using two's complement.

Integer Types in C#

In C# there are eight integer data types either **signed** or **unsigned**. Depending on the amount of bytes allocated for each type, different value ranges are determined. Here are descriptions of the types:

Type	Size	Range	Type in .NET Framework
`sbyte`	8 bits	-128 ÷ 127	`System.SByte`

byte	8 bits	0 ÷ 255	**System.Byte**
short	16 bits	-32,768 ÷ 32,767	**System.Int16**
ushort	16 bits	0 ÷ 65,535	**System.UInt16**
int	32 bits	-2,147,483,648 ÷ 2,147,483,647	**System.Int32**
uint	32 bits	0 ÷ 4,294,967,295	**System.UInt32**
long	64 bits	-9,223,372,036,854,775,808 ÷ 9,223,372,036,854,775,807	**System.Int64**
ulong	64 bits	0 ÷ 18,446,744,073,709,551,615	**System.UInt64**

We will take a brief look at the most used ones. The most commonly used integer type is **int**. It is represented as a 32-bit number with two's complement and takes a value in the range [-2^{31}, $2^{31}-1$]. Variables of this type are most frequently used to operate loops, index arrays and other integer calculations. In the following table an example of a variable of the type **int** is being declared:

```
int integerValue = 25;
int integerHexValue = 0x002A;
int y = Convert.ToInt32("1001", 2); // Converts binary to int
```

The type **long** is the largest signed integer type in C#. It has a size of 64 bits (8 bytes). When giving value to the variables of type **long** the Latin letters "**l**" or "**L**" are placed at the end of the integer literal. Placed at that position, this modifier signifies that the literal has a value of the type **long**. This is done because by default all integer literals are of the type **int**. In the next example, we declare and give 64-bit value to variables of type **long**:

```
long longValue = 9223372036854775807L;
long newLongValue = 932145699054323689l;
```

An important condition is not to exceed the range of numbers that can be represented in the used type. However, C# offers the ability to **control what happens when an overflow occurs**. This is done via the **checked** and **unchecked** blocks. The first are used when the application needs to throw an exception (of the type **System.OverflowException**) in case that the range of the variable is exceeded. The following programming code does exactly that:

```
checked
{
  int a = int.MaxValue;
  a = a + 1;
  Console.WriteLine(a);
}
```

In case the fragment is in an **unchecked** block, an exception will not be thrown and the output result will be wrong:

```
-2147483648
```

In case these blocks are not used, the C# compiler works in unchecked mode by default.

C# includes unsigned types, which can be useful when a larger range is needed for the variables in the scope of the positive numbers. Below are some examples for declaring variables without a sign. We should pay attention to the suffixes of **ulong** (all combinations of **U**, **L**, **u**, **l**).

```
byte count = 50;
ushort pixels = 62872;
uint points = 4139276850; // or 4139276850u, 4139276850U
ulong y = 18446744073709551615; // or UL, ul, Ul, uL, Lu, lU
```

Big-Endian and Little-Endian Representation

There are two ways for ordering bytes in the memory when representing integers longer than one byte:

- **Little-Endian (LE)** – bytes are ordered from left to right from the lowest-order to the highest. This representation is used in the Intel x86 and Intel x64 microprocessor architecture.
- **Big-Endian (BE)** – bytes are ordered from left to right starting with the highest-order and ending with the lowest. This representation is used in the PowerPC, SPARC and ARM microprocessor architecture.

Here is an example: the number **A8B6EA72**$_{(16)}$ is presented in both byte orders in the following way:

There are some classes in C# that offer the opportunity to define which order standard to be used. This is important for operations like sending / receiving

streams of information over the internet or other types of communication between devices made by different standards. The field **IsLittleEndian** of the **BitConverter** class for example shows what mode the class is working in and how it stores data on the current computer architecture.

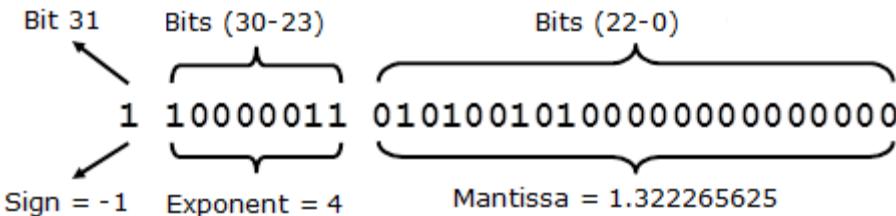
Representing Real Floating-Point Numbers

Real numbers consist of a whole and fraction parts. In computers, they are represented as **floating-point numbers**. Actually this representation comes from the Standard for Floating-Point Arithmetic (IEEE 754), adopted by the leading microprocessor manufacturers. Most hardware platforms and programming languages allow or require the calculations to be done according to the requirements of this standard. The standard defines:

- **Arithmetical formats**: a set of binary and decimal data with a floating-point, which consists of a finite number of digits.
- **Exchange formats**: encoding (bit sequences), which can be used for data exchange in an effective and compact form.
- **Rounding algorithms**: methods, which are used for rounding up numbers during calculations.
- **Operations**: arithmetic and other operations of the arithmetic formats.
- **Exceptions**: they are signals for extraordinary events such as division by zero, overflowing and others.

According to the IEEE-754 standard a random real number ***R*** can be presented in the following way:

$$R = M * q^p$$

where ***M*** is the **mantissa** of the number, ***p*** is the **order** (**exponent**), and ***q*** accordingly is the base of the numeral system the number is in. The mantissa must be a positive or negative common fraction $|M|<1$, and the exponent – a positive or negative integer.

In the mentioned method of representation of numbers, every floating-point number will have the following summarized format $\pm 0,M*q^{\pm p}$.

When notating numbers in the floating-point format using the binary numeral system in particular, we will have $R = M * 2^p$. In this representation of real numbers in the computer memory, when we change the exponent, the decimal point in the mantissa moves ("floats"). The floating-point representation format has a semi-logarithmic form. It is depicted in the following figure:

	2^{k-1}		2^0	2^{-1}	2^{-2}		2^{-n}
S	p_0	...	p_{k-1}	M_0	M_1	...	M_{n-1}
Sign	Exponent			Mantissa			

Representing Floating-Point Numbers – Example

Let's give an example of how a floating-point number is represented in the memory. We want to write the number -21.15625 in 32-bit (single precision) floating-point format according to the IEEE-754 standard. In this format, 23 bits are used for the mantissa, 8 bits for the exponent and 1 bit for the sign. The notation of the number is as follows:

The sign of the number is negative, which means that the mantissa has a negative sign:

$S = -1$

The exponent has a value of 4 (represented with a shifted order):

$p = (2^0 + 2^1 + 2^7) - 127 = (1+2+128) - 127 = 4$

For transitioning to the real value we subtract 127 from the additional code because we are working with 8 bits ($127 = 2^7-1$) starting from the zero position.

The mantissa has the following value (without taking the sign into account):

$M = 1 + 2^{-2} + 2^{-4} + 2^{-7} + 2^{-9} =$

$= 1 + 0.25 + 0.0625 + 0.0078125 + 0.001953125 =$

$= 1.322265625$

We should note that we added a one, which was missing from the binary notation of the mantissa. We did it because the mantissa is always normalized and starts with a one by default.

The value of the number is calculated using the formula $\boldsymbol{R = M * 2^p}$, which in our example looks like the following:

$R = -1{,}3222656 * 2^4 = -1{,}322265625 * 16 = -21{,}1562496 \approx -21{,}15625$

Mantissa Normalization

To use the order grid more fully, the mantissa must contain a one in its highest-power order. Every mantissa fulfilling this condition is called **normalized**. In the **IEEE-754** standard, the one in the whole part of the mantissa is by default, meaning the mantissa is always a number between 1 and 2.

If during the calculations a result that does not fulfill this condition is reached, it means that the normalization is violated. This requires the normalization of

the number prior to its further processing, and for this purpose the decimal point in the mantissa is moved and the corresponding order change is made.

The Float and Double Types in C#

In C# we have at our disposal two types, which can represent floating-point numbers. The **float** type is a 32-bit real number with a floating-point and it is accepted to be called single precision floating-point number. The **double** is a 64-bit real number with a floating-point and it is accepted that it has a double precision floating-point. These real data types and the arithmetic operations with them correspond to the specification outlined by the **IEEE 754-1985 standard**. In the following table are presented the most important characteristics of the two types:

Type	Size	Range	Significant Digits	Type in .NET Framework
float	32 bits	$\pm 1.5 \times 10^{-45} \div \pm 3.4 \times 10^{38}$	7	**System.Single**
double	64 bits	$\pm 5.0 \times 10^{-324} \div \pm 1.7 \times 10^{308}$	15-16	**System.Double**

In the **float** type we have a mantissa, which contains 7 significant digits, while in the **double** type it stores 15-16 significant digits. The remaining bits are used for specifying the sign of the mantissa and the value of the exponent. The **double** type, aside from the larger number of significant digits, also has a larger exponent, which means that it has a larger scope of the values it can assume. Here is an example how to declare variables of the **float** and **double** types:

```
float total = 5.0f;
float result = 5.0f;
double sum = 10.0;
double div = 35.4 / 3.0;
double x = 5d;
```

The suffixes placed after the numbers on the right side of the equation, serve the purpose of specifying what type the number should be treated as (**f** for **float**, **d** for **double**). In this case they are in place because by default 5.0 will be interpreted as a **double** and 5 – as an **int**.

In C#, floating-point numbers literals by default are of the double type.

Integers and floating-point numbers can both be present in a given expression. In that case, the integer variables are converted to floating-point variables and the result is defined according to the following rules:

1. If any of the floating-point types is a **double**, the result will be **double** (or **bool**).
2. If there is no **double** type in the expression, the result is **float** (or **bool**).

Many of the mathematical operations can yield results, which have no specific numerical value, like the value "+/- infinity" or **NaN** (which means "Not a Number"), these values are not numbers. Here is an example:

```
double d = 0;
Console.WriteLine(d);
Console.WriteLine(1/d);
Console.WriteLine(-1/d);
Console.WriteLine(d/d);
```

If we execute it we get the following result:

```
0.0
Infinity
-Infinity
NaN
```

If we execute the code above using **int** instead of **double**, we will receive a **System.DivideByZeroException**, because integer division by 0 is not an allowed operation.

Errors When Using Floating-Point Numbers

Floating-point numbers (presented according to the IEEE 754 standard) are very convenient for calculations in physics, where very big numbers are used (with several hundred digits) and also numbers that are very close to zero (with hundreds of digits after the decimal point before the first significant digit). When working with these numbers, the **IEEE 754 format** is exceptionally convenient because it keeps the number's order in the exponent and the mantissa is only used to store the significant digits. In 64-bit floating-point numbers accuracy of 15-16 digits, as well as exponents displacing the decimal point with 300 positions left or right can be achieved.

Unfortunately **not every real number has an exact representation in the IEEE 754 format**, because not each number can be presented as a polynomial of a finite number of addends, which are negative powers of two. This is fully valid even for numbers, which are used daily for the simplest financial calculations. For example the number 0.1 represented as a 32-bit floating-point value is presented as 0.099999994. If the appropriate rounding is used, the number can be accepted as 0.1, but the error can be accumulated and cause serious deviations, especially in financial calculations. For example when adding up 1000 items with a unit price of 0.1 EUR each, we should get a sum of 100 EUR but if we use a 32-bit floating-point numbers for the

calculations the result will be 99.99905. Here is C# example in action, which proves the errors caused by the inaccurate presentation of decimal real numbers in the binary numeral system:

```
float sum = 0f;
for (int i = 0; i < 1000; i++)
{
  sum += 0.1f;
}
Console.WriteLine("Sum = {0}", sum);
// Sum = 99.99905
```

We can easily see the errors in such calculations if we execute the example or modify it to get even more striking errors.

Precision of Floating-Point Numbers

The accuracy of the results from floating-point calculations depends on the following parameters:

1. Precision of the number representation.
2. Precision of the used number methods.
3. Value of the errors resulting from rounding up, etc.

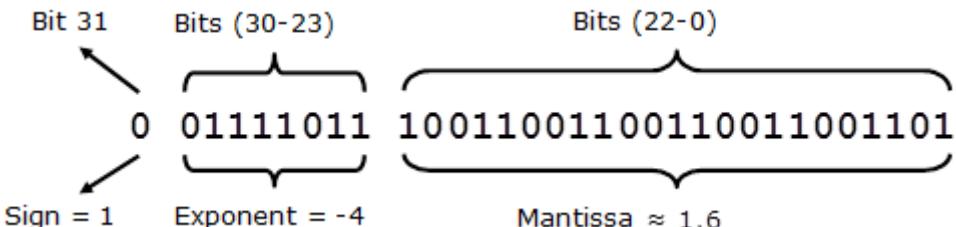
Calculations with them can be inaccurate because they are represented in the memory with some kind of precision. Let's look at the following code fragment as an example:

```
double sum = 0.0;
for (int i = 1; i <= 10; i++)
{
  sum += 0.1;
}
Console.WriteLine("{0:r}", sum);
Console.WriteLine(sum);
```

During the execution, in the loop we add the value 1/10 to the variable **sum**. When calling the **WriteLine()** method, we use the round-trip format specifier **"{0:r}"** to print the exact (not rounded) value contained in the variable, and after that we print the same value without specifying a format. We expect that when we execute the program we will get 1.0 as a result but in reality, when rounding is turned off, the program returns a value very close to the correct one but still different:

```
0.99999999999999989
1
```

As we can see in the example, by default, when printing floating-point numbers in .NET Framework, **they are rounded**, which seemingly reduces the errors of their inaccurate notation in the IEEE 754 format. The result of the calculation above is obviously wrong but after the rounding it looks correct. However, if we add 0.1 a several thousand times, the error will accumulate and the rounding will not be able to compensate it.

The reason for the wrong answer in the example is that the number 0.1 does not have an exact representation in the **double** type and it has to be rounded. Let's replace **double** with **float**:

```
float sum = 0.0f;
for (int i = 1; i <= 10; i++)
{
   sum += 0.1f;
}
Console.WriteLine("{0:r}", sum);
```

If we execute the code above, we will get an entirely different sum:

```
1.00000012
```

Again the reason for this is rounding.

If we investigate why the program yields these results, we will see that the number 0.1 of the **float** type is represented in the following manner:

All this looks correct except for the mantissa, which has a value slightly bigger than 1.6, not exactly 1.6 because this number cannot be presented as sum of the negative powers of 2. If we have to be very precise, the value of the mantissa is 1 + 1 / 2 + 1 / 16 + 1 / 32 + 1 / 256 + 1 / 512 + 1 / 4096 + 1 / 8192 + 1 / 65536 + 1 / 131072 + 1 / 1048576 + 1 / 2097152 + 1 / 8388608 ≈ 1.6000000238418579101562 5 ≈ 1.6. Thus the number 0.1 presented in the IEE 754 is slightly more than 1.6×2^{-4} and the error occurs not during the addition but before that, when 0.1 is recorded in the **float** type.

Double and **Float** types have a field called **Epsilon**, which is a constant, and it contains the smallest value larger than zero, which can be represented by an instance of **System.Single** or **System.Double** respectively. Each value smaller than **Epsilon** is considered to be equal to 0. For example, if we

compare two numbers, which are different after all, but their difference is smaller than **Epsilon**, they will be considered equal.

The Decimal Type

The **System.Decimal** type in .NET Framework uses **decimal floating-point arithmetic** and 128-bit precision, which is very suitable for big numbers and precise financial calculations. Here are some characteristics of the **decimal** type:

Type	Size	Range	Significant numbers	Type in .NET framework
decimal	128 bits	$\pm1.0 \times 10^{-28} \div \pm7.9 \times 10^{28}$	28-29	**System.Decimal**

Unlike the floating-point numbers, the **decimal** type **retains its precision** for all decimal number in its range. The secret to this excellent precision when working with decimal numbers lies in the fact that the internal representation of the mantissa is not in the binary system but in the decimal one. The exponent is also a power of 10, not 2. This enables numbers to be represented precisely, without them being converted to the binary numeral system.

Because the **float** and **double** types and the operations on them are implementer by the **arithmetic coprocessor**, which is part of all modern computer microprocessors, and **decimal** is implemented by the software in .NET CLR, it is tens of times slower than **double**, but is irreplaceable for the execution of financial calculations.

In case our target is to assign a given literal to variable of type **decimal**, we need to use the suffixes **m** or **M**. For example:

```
decimal calc = 20.4m;
decimal result = 5.0M;
```

Let's use **decimal** instead of **float** / **double** in the example from before:

```
decimal sum = 0.0m;
for (int i = 1; i <= 10000000; i++)
{
  sum += 0.0000001m;
}
Console.WriteLine(sum);
```

This time the result is exactly what we expected:

```
1.0000000
```

Even though the **decimal** type has a higher precision than the floating-point types, it has a smaller value range and, for example, it cannot be used to represent the following value **1e-50**. As a result, an overflow may occur when converting from floating-point numbers to **decimal**.

Character Data (Strings)

Character (text) data in computing is text, encoded using a sequence of bytes. There are different encoding schemes used to encode text data. Most of them encode one character in one byte or in a sequence of several bytes. Such encoding schemes are ASCII, Windows-1251, UTF-8 and UTF-16.

Encoding Schemes (Encodings)

The **ASCII** encoding scheme compares the unique number of the letters from the Latin alphabet and some other symbols and special characters and writes them in a single byte. The ASCII standard contains a total of 127 characters, each of which is written in one byte. A text, written as a sequence of bytes according to the ASCII standard, cannot contain Cyrillic or characters from other alphabets such as the Arabian, Korean and Chinese ones.

Like the ASCII standard, the **Windows-1251** encoding scheme compares the unique number of the letters in the Latin alphabet, Cyrillic and some other symbols and specialized characters and writes them in one byte. The Windows-1251 encoding defines the numbers of 256 characters – exactly as many as the different values that can be written in one byte. A text written according to the Windows-1251 standard can contain only Cyrillic and Latin letters, Arabian, Indian or Chinese are not supported.

The **UTF-8** encoding is completely different. All characters in the Unicode standard – the letters and symbols used in all widely spread languages in the world (Cyrillic, Latin, Arabian, Chinese, Japanese, Korean and many other languages and writing systems) – can be encoded in it. The UTF-8 encoding contains over half a million symbols. In the UTF-8 encoding, the more commonly used symbols are encoded in 1 byte (Latin letters and digits for example), the second most commonly used symbols are coded in 2 bytes (Cyrillic letters for example), and the ones that are used even more rarely are coded in 3 or 4 bytes (like the Chinese, Japanese and Korean alphabet).

The **UTF-16** encoding, like UTF-8 can depict text of all commonly used languages and writing systems, described in the Unicode standard. In UTF-16, every symbol is written in 16 bits (2 bytes) and some of the more rarely used symbols are presented as a sequence of two 16-bit values.

Presenting a Sequence of Characters

Character sequences can be presented in several ways. The most common method for writing text in the memory is to write in 2 or 4 bytes its length, followed by a sequence of bytes, which presents the text itself in some sort of encoding (for example Windows-1251 or UTF-8).

Another, less common method of writing texts in the memory, typical for the C language, represents texts as a sequence of characters, usually coded in 1 byte, followed by a special ending character, most frequently a 0. When using this method, the length of the text saved at a given position in the memory is not known in advance. This is considered a disadvantage in many situations.

Char Type

The **char** type in the C# language is a **16-bit value**, in which a single **Unicode character** or part of it is coded. In most alphabets (for example the ones used by all European languages) one letter is written in a single 16-bit value, and thus it is assumed that a variable of the **char** type represents a single character. Here is an example:

```
char ch = 'A';
Console.WriteLine(ch);
```

String Type

The **string** type in C# **holds text, encoded in UTF-16**. A single string in C# consists of 4 bytes length and a sequence of characters written as 16-bit values of the **char** type. The **string** type can store texts written in all widespread alphabets and human writing systems – Latin, Cyrillic, Chinese, Japanese, Arabian and many, many others. Here is an example of the usage of the **string**:

```
string str = "Example";
Console.WriteLine(str);
```

Exercises

1. Convert the numbers **151**, **35**, **43**, **251**, **1023** and **1024** to the **binary numeral system**.
2. Convert the number **1111010110011110**$_{(2)}$ to **hexadecimal** and **decimal** numeral systems.
3. Convert the hexadecimal numbers **FA**, **2A3E**, **FFFF**, **5A0E9** to **binary** and **decimal** numeral systems.
4. Write a program that converts a **decimal number to binary** one.
5. Write a program that converts a **binary number to decimal** one.
6. Write a program that converts a **decimal number to hexadecimal** one.
7. Write a program that converts a **hexadecimal number to decimal** one.
8. Write a program that converts a **hexadecimal number to binary** one.
9. Write a program that converts a **binary number to hexadecimal** one.

10. Write a program that converts a **binary number to decimal** using the Horner scheme.
11. Write a program that converts **Roman digits to Arabic** ones.
12. Write a program that converts **Arabic digits to Roman** ones.
13. Write a program that by given **N**, **S**, **D** (**2** ≤ **S**, **D** ≤ **16**) converts the number **N** from an **S**-based numeral system to a **D** based numeral system.
14. Try **adding up 50,000,000 times the number 0.000001**. Use a loop and addition (not direct multiplication). Try it with **float** and **double** and after that with **decimal**. Do you notice the **huge difference in the results** and speed of calculation? Explain what happens.
15. * Write a program that prints the value of the **mantissa**, the **sign of the mantissa** and **exponent** in **float** numbers (32-bit numbers with a floating-point according to the **IEEE 754** standard). Example: for the number **-27.25** should be printed: **sign** = **1**, **exponent** = **10000011**, **mantissa** = **10110100000000000000000**.

Solutions and Guidelines

1. Use the **methods for [conversion from one numeral system to another](#)**. You can check your results with the help of the Windows built-in calculator, which supports numeral systems in **"Programmer" mode**. The results are: **10010111**, **100011**, **101011**, **11111011**, **1111111111** and **10000000000**.
2. Like the previous exercise. Result: $\mathbf{F59E_{(16)}}$ and $\mathbf{62878_{(10)}}$.
3. Like the previous exercise. The results are: $\mathbf{FA_{(16)}}$ = $\mathbf{250_{(10)}}$ = $\mathbf{11111010_{(2)}}$, $\mathbf{2A3E_{(16)}}$ = $\mathbf{10814_{(10)}}$ = $\mathbf{10101000111110_{(2)}}$, $\mathbf{FFFF_{(16)}}$ = $\mathbf{65535_{(10)}}$ = $\mathbf{1111111111111111_{(2)}}$ and $\mathbf{5A0E9_{(16)}}$ = $\mathbf{368873_{(10)}}$ = $\mathbf{1011010000011101001_{(2)}}$.
4. The rule is "**divide by 2 and concatenate the remainders in reversed order**". For division with a remainder we use the **%** operator. You can cheat by invoking **Convert.ToString(numDecimal, 2)**.
5. Start with a **sum of 0**. Multiply the **right-most bit** with **1** and add it to the sum. Multiply the **next bit** on the left by **2** and add it to the sum. Multiply the **next bit** on the left by **4**, the **next** by **8** and so on. You can cheat by invoking **Convert.ToInt32(binaryNumAsString, 2)**.
6. The rule is "**divide by the base of the system (16) and concatenate the remainders in reversed order**". A logic that gets a hexadecimal digit (**0**…**F**) by decimal number (**0**…**15**) should also be implemented. You can cheat by invoking **num.ToString("X")**.
7. Start with a **sum of 0**. Multiply the **right-most digit** with **1** and add it to the sum. Multiply the **next digit** to the left by **16** and add it to the sum.

Multiply the **next digit** by **16*16**, the **next** by **16*16*16** and so on. You can cheat by invoking **Convert.ToInt32(hexNumAsString, 16)**.

8. Use the fast method for transitioning between hexadecimal and binary numeral system (**each hexadecimal digit turns to 4 binary bits**).

9. Use the fast method for transitioning from binary to hexadecimal numeral system (**each 4 binary bits correspond to a hexadecimal digit**).

10. Directly apply the Horner scheme.

11. **Scan the digits of the Roman number** from left to right and add them up to a sum, which is initialized with a 0. When processing each Roman digit, take it with a positive or negative sign, **depending on the digit after it** (whether it has a bigger or smaller decimal value).

12. Take a look at the numbers from **1 to 9** and their corresponding Roman representation with the digits **"I", "V" and "X"**:

    1 -> I
    2 -> II
    3 -> III
    4 -> IV
    5 -> V
    6 -> VI
    7 -> VII
    8 -> VIII
    9 -> IX

    We have exactly the same correspondence for the numbers **10, 20, ..., 90** with their Roman representation **"X", "L" and "C"**. The same is valid for the numbers **100, 200, ..., 900** and their Roman representation with **"C", "D" and "M"** and so on.

    We are now ready to **convert the number N into the Roman numeral system**. It must be in the range [1…3999], otherwise we should report an error. First we separate the thousands (N / 1000) and replace them with their Roman counterpart. After that we separate the hundreds (N / 100) % 10) and separate them with their Roman counterpart and so on.

13. You can convert first from **S-based system** to **decimal number** and then from decimal number to **D-based system**.

14. If you execute the calculations correctly, you will get **32.00** (for **float**), **49.9999999657788** (for **double**) and **50.00** (for **decimal**) respectively. The differences come from the fact that **0.000001** has no exact representation as **float** and **double**. You may notice also that adding **decimal** values is at least **10 times slower** than adding **double** values.

15. Use the special method for conversion of single precision floating-point numbers to a sequence of 4 bytes: **System.BitConverter.GetBytes(<float>)**. Then use **bitwise operations** (shifting and bit masks) to extract the sign, mantissa and exponent following the IEEE 754 standard.

Chapter 9. Methods

In This Chapter

In this chapter we will get more familiar with what **methods** are and why we need to use them. The reader will be shown how to **declare methods**, what **parameters** are and what a method's signature is, how to **call a method**, how to **pass arguments** of methods and how methods **return values**. At the end of this chapter we will know how to create our own method and how to use (invoke) it whenever necessary. Eventually, we will suggest some good practices in working with methods. The content of this chapter accompanied by detailed examples and exercises that will help the reader practice the learned material.

Subroutines in Programming

To solve a certain task, especially if it is a complex one, we apply the method that ancient Romans did "**divide and conquer**". According to this principle, the problem we solve must be divided into small subproblems. Taken separately they are well defined and easy to be resolved compared to the original problem. At the end by finding solutions for all the small problems we solve the complex one.

Using the same analogy, whenever we write a software program we aim to solve particular task. To do it in an efficient and "easy-to-make" way we use the same mentioned above principle "divide and conquer". We separate the given task into smaller tasks, then develop solutions for them and put them together into one program. Those smaller tasks we call **subroutines**.

In some other programming languages subroutines can be named as functions or procedures. In C#, they are called methods.

What Is a "Method"?

A **method** is a basic part of a program. It can **solve a certain problem, eventually take parameters and return a result**.

A method represents all data conversion a program does, to resolve a particular task. Methods consist of the **program's logic**. Moreover they are the place where the "real job" is done. That is why methods can be taken as a base unit for the whole program. This on the other hand, gives us the opportunity, by using a simple block, to build bigger programs, which resolve more complex and sophisticated problems. Below is a simple example of a method that calculates rectangle's area:

```
static double GetRectangleArea(double width, double height)
{
  double area = width * height;
  return area;
}
```

Why to Use Methods?

There are many reasons we should use methods. Some of them are listed below, and by gaining experience, you will assure yourself that methods are something that cannot be avoided for a serious task.

Better Structured Program and More Readable Code

Whenever a program has been created, it is always a good practice to use methods, in a way to **make your code better structured and easy to read**, hence to be maintained by other people.

A good reason for this is the fact, that of the time that a program exists, only about 20% of the effort is spent on creating and testing the program. The rest is for **maintenance** and adding new features to the initial version. In most of the cases, once the code has been released, it is maintained not only from its creator, but by many other developers. That is why it is very important for the code to be as well structured and readable as possible.

Avoid Duplicated Code

Another very important reason to use methods is that methods help us to **avoid code repeating**. This has a strong relationship to the idea of **code reuse**.

Code Reuse

If a piece of code is used more than once in a program, it is good to separate it in a method, which can be called many times – thus enabling reuse of the same code, without rewriting it. This way we **avoid code repeating**, but this is not the only advantage. The program itself becomes more readable and **well structured**.

Repeating code may become very noxious and hazardous, because it impedes the maintenance of the program and leads to errors. Often, whenever change of repeating code is needed, the developer fixes only some of the blocks, but the problems is still alive in the others, about which they forgot. So for example if a defect is found into a piece of 50 lines code, that is copied to 10 different places over the program, to fix the defect, the repeated code must be fixed for the all 10 places. This, however, is not what really happens. Often, due to lack of concentration or some other reasons, the developer **fixes only some of the pieces of code, but not all of them**. For example,

let's say that in our case the developer has fixed 8 out of 10 blocks of code. This eventually, will lead to unexpected behavior of our program, only in rare cases and, moreover, it will be very a difficult task to find out what is going wrong with the program.

How to Declare, Implement and Invoke a Method?

This is the time to learn how to distinguish three different actions related to existing of a method: declaring, implementation (creation) and calling of a method.

Declaring a method we call method registration in the program, so it can be successfully identified in the rest of the program.

Implementation (creation) of a method is the process of typing the code that resolves a particular task. This code is in the method itself and represents its logic.

Method call is the process that invokes the already declared method, from a part of the code, where a problem, that the method resolves, must be solved.

Declaring Our Own Method

Before we learn how to declare our own method, it is important to know where we are allowed to do it.

Where Is Method Declaration Allowed?

Although we still haven't explained how to declare a class, we have seen it in the exercises before. We know that every class has opening and closing curly brackets – "**{**" and "**}**", between which the program code is placed. More detailed description for this can be found in the chapter "Defining Classes", however we mention it here, because a method exists only if it is declared **between the opening and closing brackets of a class** – "**{**" and "**}**". In addition a method **cannot** be declared inside another method's body (this will be clarified later).

In the C# language, a method can be declared only between the opening "{" and the closing "}" brackets of a class.

A typical example for a method is the already known method **Main(…)** – that is always declared between the opening and the closing curly brackets of our class. An example for this is shown below:

HelloCSharp.cs

```
public class HelloCSharp
{ // Opening brace of the class
```

```
    // Declaring our method between the class' body braces
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C#!");
    }
} // Closing brace of the class
```

Method Declaration

To **declare a method** means to **register** the method in our program. This is shown with the following declaration:

```
[static] <return_type> <method_name>([<param_list>])
```

There are some mandatory elements to declare method:

- Type of the result, returned by the method – **<return_type>**.
- Method's name – **<method_name>**.
- List of parameters to the method – **<param_list>** – it can be empty list or it can consist of a sequence of parameters declarations.

To clarify the **elements of method's declaration**, we can use the **Main(…)** method from the example **HelloCSharp** show in the previous block:

```
static void Main(string[] args)
```

As can be seen the **type of returned value** is **void** (i.e. that method does not return a result), the method's name is **Main**, followed by round brackets, between which is a list with the method's **parameters**. In the particular example it is actually only one **parameter** – the array **string[] args**.

The sequence, in which the elements of a method are written, is strictly defined. Always, at the very first place, is the type of the value that method returns **<return_type>**, followed by the method's name **<method_name>** and list of parameters at the end **<param_list>** placed between in round brackets – "**(**" and "**)**". Optionally the declarations can have **access modifiers** (as **public** and **static**).

When a method is declared keep the sequence of its elements description: first is the type of the value that the method returns, then is the method's name, and at the end is a list of parameters placed in round brackets.

The list with parameters is allowed to be **void** (empty). In that case the only thing we have to do is to type "**()**" after the method's name. Although the

method has not parameters the round brackets must follow its name in the declaration.

	The round brackets – "(" and ")", are always placed after the method's name, no matter whether it has or has not any parameters.

For now we will not focus at what **<return_type>** is. For now we will use **void**, which means the method will not return anything. Later, we will see how that can be changed

The keyword **static** in the description of the declaration above is not mandatory but should be used in small simple programs. It has a special purpose that will be explained later in this chapter. Now the methods that we will use for example, will include the keyword **static** in their declaration. More about methods that are not declared as **static** will be discussed in the chapter "Defining Classes", section "Static Members".

Method Signature

Before we go on with the basic elements from the method's declaration, we must pay attention to something more important. In object-oriented programming a method is identified by a pair of elements of its declaration: name of the method, and list of parameters. These two elements define the so-called **method specification** (often can be found as a **method signature**).

C#, as a language used for object oriented programming, also distinguishes the methods using their specification (signature) – method's name **<method_name>** and the list with parameters – **<param_list>**.

Here we must note that the type of returned value of a method is only part of its declaration, not of its signature.

	What identifies a method is its signature. The return type is not part of the method signature. The reason is that if two methods differ only by their return value types, for the program is not clear enough which of them must be called.

A more detailed explanation on why the type of the returned value is not part of the method signature, you will find later in this chapter.

Method Names

Every method solves a particular task from the whole problem that our program solves. **Method's name** is used when method is called. Whenever we call (start) a particular method, we type its name and if necessary we pass values (if there are any).

In the example below, the name of our method is **PrintLogo**:

```
static void PrintLogo()
{
  Console.WriteLine("Microsoft");
  Console.WriteLine("www.microsoft.com");
}
```

Rules to Name a Method

It is recommended, when declare a method, to follow **the rules for method naming** suggested by Microsoft:

- The name of a method must start with **capital letter**.
- The **PascalCase** rule must be applied, i.e. each new word, that concatenates so to form the method name, must start with capital letter.
- It is recommended that the method name must consist of **verb, or verb and noun**.

Note that these rules are not mandatory, but recommendable. If we aim our C# code to follow the style of all good programmers over the globe, we must use Microsoft's code convention. A more detailed recommendation about method naming will be given in the chapter "High-Quality Code", section "Naming Methods".

Here some examples for well named methods:

```
Print
GetName
PlayMusic
SetUserName
```

And some examples for bad named methods:

```
Abc11
Yellow___Black
foo
_Bar
```

It is very important that the method name describes the method's purpose. All behind this idea is that when a person that is not familiar with our program reads the method name, they can easily understand what that method does, without the need to look at the method's source code.

To name a method it is good to follow these rules:

- **Method name must describe the method's purpose.**
- **Method name must begin with capital letter.**

	- **The `PascalCase` rule must be applied.** - **The method name must consist of verb, or verb and noun.**

Modifiers

A **modifier** is a keyword in C#, which gives additional information to the compiler for a certain code.

We have already met some modifiers – **`public`** and **`static`**. Now we will briefly describe what **modifiers** are actually. Detailed description will be given later in the chapter "Defining Classes", section "Access Modifiers". So let's begin with an example:

```
public static void PrintLogo()
{
  Console.WriteLine("Microsoft");
  Console.WriteLine("www.microsoft.com");
}
```

With this example we define a public method by the modifier **`public`**. It is a special type modifier, called also **access modifier** and is used to show that method can be called by any C# class, no matter where it is. Public modifiers are not restricted in the meaning of "who" can call them.

Another example for access modifier, that we can meet, is the modifier **`private`**. Its function is opposite to that of the **`public`**, i.e. if a method is declared by access modifier **`private`**, it cannot be called from anywhere, except from the class in which it is declared.

If a method is declared **without an access modifier** (either **`public`** or **`private`**), it is **accessible from all classes** in the current assembly, but not accessible for any other assemblies (let say from other projects in Visual Studio). For the same reason, when we are writing small programs, like those in this chapter, we will not specify access modifiers.

For now, the only thing that has to be learned is that in method declaration there cannot be more than one access modifier.

When a method has a keyword **`static`**, in its declaration, this method is called **static**. To call a static method there is no need to have an instance of a class in which the static method is declared. For now the reader can accept that, the methods must be static. Dealing with non-static methods will be explained in the chapter "Defining Classes", section "Methods".

Implementation (Creation) of Own Method

After a method had been declared, we must write its implementation. As we already explained above, **implementation (body)** of the method consists of the code, which will be executed by calling the method. That code must be placed in the method's body and it represents the method's logic.

The Body of a Method

Method body we call the piece of code, that is placed in between the curly brackets "**{**" and "**}**", that directly follow the method declaration.

```
static <return_type> <method_name>(<parameters_list>)
{
  // … code goes here – in the method's body …
}
```

The real job, done by the method, is placed exactly in the method body. So, the algorithm used in the method to solve the particular task is placed in the method body.

So far we have seen many examples of method body however, we will show one more with the code below:

```
static void PrintLogo()
{ // Method's body starts here
  Console.WriteLine("Microsoft");
  Console.WriteLine("www.microsoft.com");
} // … And finishes here
```

Let's consider one more time one rule about method declaration:

Method can NOT be declared inside the body of another method.

Local Variables

Whenever we declare a variable inside the body of a method, we call that variable **local variable** for the method. To name a variable we should follow the identifiers rules in C# (refer to chapter "Primitive Types and Variables").

The area where a local variable exists, and can be used, begins from the line where the variable is declared and ends at the closing curly bracket "**}**" of the method body. This is the so-called **area of visibility of the variable** (**variable scope**). If we try to declare variable, after we have already declared a variable with the same name, the code will not compile due to an error. Let's look at the example below:

```
static void Main()
{
  int x = 3;
  int x = 4;
}
```

Compiler will not let's use the name **x** for two different variables, and will return a message similar to the one below:

```
A local variable named 'x' is already defined in this scope.
```

A **block of code** we call a code that is placed between opening and closing curly brackets "**{**" and "**}**".

If a variable is declared within a block, it is also called **local** (for this block). Its area of visibility begins from the line where the variable is declared, and ends at the line where block's closing bracket is.

Invoking a Method

Invoking or **calling a method** is actually the process of **execution** of the method's code, placed into its body.

It is very easy to invoke a method. The only thing that has to be done is to write the method's name **<method_name>**, followed by the round brackets and semicolon "**;**" at the end:

```
<method_name>();
```

Later will see an example for when the invoked method has a parameter list (in the case here the method has no parameters).

To clarify how method invocation works, the next fragment shows how the method **PrintLogo()** will be called:

```
PrintLogo();
```

Result of method's execution is:

```
Microsoft
www.microsoft.com
```

Who Takes Control over the Program when We Invoke a Method?

When a method executes it **takes control over the program**. If in the caller method, however, we call another one, the caller will give the control to the called method. The called method will return back the control to the caller

right after its execution finishes. The execution of the caller will continue from that line, where it was before calling the other method.

For example, let's call **PrintLogo()** from the **Main()** method:

```
class MethodControlTest
{
  static void PrintLogo()
  {
    Console.WriteLine("Microsoft");
    Console.WriteLine("www.microsoft.com");
  }

  static void Main()
  {
    // ... Some code here ...

    PrintLogo();

    // ... Some code here ...
  }
}
```

First the code of method **Main()**, that is marked with (1) will be executed, then the control of the program will be given to the method **PrintLogo()** – the dotted arrow (2). This will cause the execution of the code in method **PrintLogo()**, numbered with (3). When the method **PrintLogo()** work is done, the control over the program is returned back to the method **Main()** – dotted arrow (4). Execution of **Main()** will continue from the line after **PrintLogo()** call – marked with (5).

Where a Method Can Be Invoked From?

A method can be invoked from the following places:

- From the main program method – **Main()**:

```
static void Main()
{
  PrintLogo();
}
```

- From some other method:

```
static void PrintLogo()
{
  Console.WriteLine("Microsoft");
```

```
    Console.WriteLine("www.microsoft.com");
}

static void PrintCompanyInformation()
{
    // Invoking the PrintLogo() method
    PrintLogo();

    Console.WriteLine("Address: One, Microsoft Way");
}
```

- A method can be invoked from its own body. Such a call is referred to as **recursion**. We will discuss it in details in the chapter "Recursion".

Method Declaration and Method Invocation

In C# the order of the methods in the class is not important. We are allowed to invoke (call) a method before it is declared in code:

```
static void Main()
{
    // …
    PrintLogo();
    // …
}

static void PrintLogo()
{
    Console.WriteLine("Microsoft");
    Console.WriteLine("www.microsoft.com");
}
```

If we create a class that contains the code above, we will see that the code will compile and run successfully. It doesn't matter whether we declared the method before or after the main method. In some other languages (like Pascal), invocation of a method that is declared below the line of the invocation is not allowed.

If a method is called in the same class, where it is declared and implemented, it can be called at a line before the line at which it is declared.

Parameters in Methods

Often to solve certain problem, the method may need additional information, which depends on the environment in what the method executes.

So if there is a method, that has to find the area of a square, in its body there must be the algorithm that finds that area (equation $S = a^2$). Since the area depends on the square side length, to calculate that equation for each square, the method will need to pass a value for the square side length. That is why we have to pass somehow that value, and for this purpose we use **parameters**.

Declaring Methods with Parameters

To pass information necessary for our method we use the **parameters list**. As was already mentioned, we must place it between the brackets following the method name, in method the declaration:

```
static <return_type> <method_name>(<parameters_list>)
{
  // Method's body
}
```

The parameters list **<parameters_list>** is a list with zero or more **declarations of variables**, separated by a comma, so that they will be used for the implementation of the method's logic:

```
<parameters_list> = [<type1> <name1>[, <typei> <namei>]],
where i = 2, 3, …
```

When we create a method, and we need certain information to develop the particular algorithm, we choose that variable from the list, which is of type **<type$_i$>** and so we use it by its name **<name$_i$>**.

The parameters from the list can be of any type. They can be primitive types (**int**, **double**, …) or object types (for example **string** or array – **int[]**, **double[]**, **string[]**, …).

Method to Display a Company Logo – Example

To make the mentioned above more clear, we will change the example that shows the logo of "Microsoft":

```
static void PrintLogo(string logo)
{
  Console.WriteLine(logo);
}
```

Now, executing our method, we can display the logo of other companies, not only of "Microsoft". This is possible because we used a parameter of type **string** to pass the company name. The example shows how to use the information given in the parameters list – the variable **logo**, which is defined

in the parameters list, is used in the method's body by the name given in the definition.

Method to Calculate the Sum of Prices of Books – Example

We mentioned above, that whenever it is necessary we can use arrays as parameters for a certain method (**int[]**, **double[]**, **string[]**, …). So let's take a look at another example to illustrate this.

Imagine we are in a bookstore and we want to calculate the amount of money we must pay for all the books we bought. We will create a method that gets the prices of all the books as an array of type **decimal[]**, and then returns the total amount we must pay:

```
static void PrintTotalAmountForBooks(decimal[] prices)
{
   decimal totalAmount = 0;
   foreach (decimal singleBookPrice in prices)
   {
      totalAmount += singleBookPrice;
   }
   Console.WriteLine("The total amount for all books is:" +
      totalAmount);
}
```

Method Behavior According to Its Input

When a method with parameters is declared, our purpose is that every time we invoke the method, its result changes according to its input. Said with another word, the algorithm is the same, but due to **input** change, the **result** changes too.

When a method has parameters, its behavior depends upon parameters values.

Method to Show whether a Number is Positive – Example

To clarify the way method execution depends upon its input let's take look at another example. The method gets as input a number of type **int**, and according to it returns to the console "**Positive**", "**Negative**" or "**Zero**":

```
static void PrintSign(int number)
{
   if (number > 0)
   {
      Console.WriteLine("Positive");
   }
   else if (number < 0)
```

```
	{
		Console.WriteLine("Negative");
	}
	else
	{
		Console.WriteLine("Zero");
	}
}
```

Method with Multiple Parameters

So far we had some examples for methods with parameter lists that consist of a **single parameter**. When a method is declared, however, it can have as **multiple parameters** as the method needs.

If we are asking for maximal of two values, for example, the method needs two parameters:

```
static void PrintMax(float number1, float number2)
{
	float max = number1;

	if (number2 > max)
	{
		max = number2;
	}
	Console.WriteLine("Maximal number: " + max);
}
```

Difference in Declaration of Methods with Multiple Parameters

When a method with multiple parameters is declared, we must note that even if the parameters are of the same type, usage of short way of variable declaration is not allowed. So the line below in the methods declaration is invalid and will produce compiler error:

```
float var1, var2;
```

Type of the parameters has to be explicitly written before each parameter, no matter if some of its neighbors are of the same type.

Hence, declaration like one shown below is not valid:

```
static void PrintMax(float var1, var2)
```

Correct way to do so is:

```
static void PrintMax(float var1, float var2)
```

Invoking Methods with Parameters

Invocation of a method with one or several parameters is done in **the same way as invocation of methods without parameters**. The difference is that between the brackets following the method name, we place values. These values (called **arguments**) will be assigned to the appropriate parameters form the declaration and will be used when method is executed.

Several examples for methods with parameters are show below:

```
PrintSign(-5);
PrintSign(balance);

PrintMax(100.0f, 200.0f);
```

Difference between Parameters and Arguments of a Method

Before we continue with this chapter, we must learn how to distinguish between parameters naming in the parameters list in the methods declaration and the values that we pass when invoking a method.

To clarify, when we declare a method, any of the elements from the parameters list we will call **parameters** (in other literature sources they can be named as **formal parameters**).

When we call a method the values we use to assign to its parameters are named as **arguments**.

In other words, the elements in the parameters list (**var1** and **varr2**) are called **parameters**:

```
static void PrintMax(float var1, float var2)
```

Accordingly, the values by the method invocation (**-23.5** and **100**) are called **arguments**:

```
PrintMax(100.0f, -23.5f);
```

Passing Arguments of a Primitive Type

As just was explained, in C# when a variable is passed as a method argument, its value is copied to the parameter from the declaration of the method. After that, the copy will be used in the method body.

There is, however, one thing we should be aware of. If the declared parameter is of a **primitive type**, the usage of the arguments does not

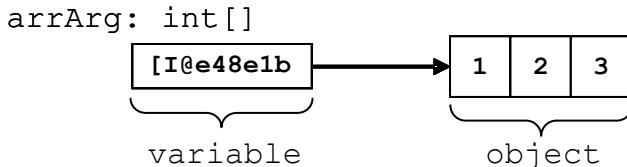
change the argument itself, i.e. the argument value will not change for the code after the method has been invoked.

So if we have piece of code like that below:

```
static void PrintNumber(int numberParam)
{
   // Modifying the primitive-type parameter
   numberParam = 5;

   Console.WriteLine("in PrintNumber() method, after " +
      "modification, numberParam is: {0}", numberParam);
}
```

Invocation of the method from **Main()**:

```
static void Main()
{
   int numberArg = 3;

   // Copying the value 3 of the argument numberArg to the
   // parameter numberParam
   PrintNumber(numberArg);

   Console.WriteLine("in the Main() method numberArg is: " +
      numberArg);
}
```

The value **3** of **numberArg**, is copied into the parameter **numberParam**. After the method **PrintNumber()** is invoked, to **numberParam** is assigned value **5**. This does not affect the value of variable **numberArg**, because by invocation of that method, the variable **numberParam** keeps a **copy** of the argument value. That is why the method **PrintNumber()** prints the number **5**. Hence, after invocation of method **PrintNumber()** in the method **Main()** what is printed is the value of **numberArg** and as it can be seen that value is not changed. The result from the above line is printed below:

```
in PrintNumber() method, after modification, numberParam is: 5
in the Main() method numberArg is: 3
```

Passing Arguments of Reference Type

When we need to declare (and so to invoke) a method, that has parameters of **reference type** (as arrays), we must be very careful.

Before explaining the reason for the above consideration, we have to remind ourselves something from chapter "Arrays". An array, as any other reference

type, consists of a variable-pointer (**object reference**) and a **value** – the real information kept in the computer's memory (we call it an **object**). In our case the object is the real array of elements. The address of this object, however, is kept in the variable (i.e. the address where the array elements are placed in the memory):

So whenever we operate with arrays in C#, we always access them by that variable (the address / pointer / reference) we used to declare the particular array. This is the principle for any other reference type. Hence, whenever an argument of a reference type is passed to a method, the method's parameter receives the reference itself. But what happens with the object then (the real array)? Is it also copied or no?

To explain this, let's have the following example: assume we have method **ModifyArray()**, that modifies the first element of an array that is passed as a parameter, so it is reinitialized the first element with value **5** and then prints the elements of the array, surrounded by square brackets and separated by commas:

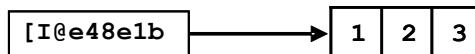
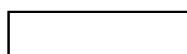
```
static void ModifyArray(int[] arrParam)
{
   arrParam[0] = 5;
   Console.Write("In ModifyArray() the param is: ");
   PrintArray(arrParam);
}

static void PrintArray(int[] arrParam)
{
   Console.Write("[");
   int length = arrParam.Length;

   if (length > 0)
   {
      Console.Write(arrParam[0].ToString());
      for (int i = 1; i < length; i++)
      {
         Console.Write(", {0}", arrParam[i]);
      }
   }
   Console.WriteLine("]");
}
```

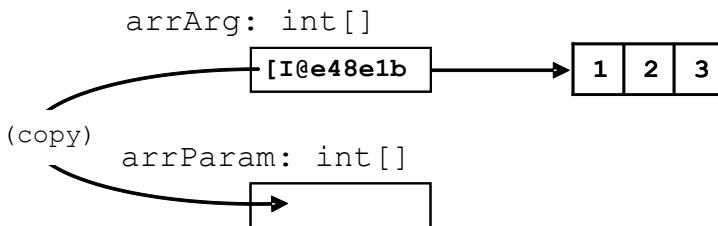
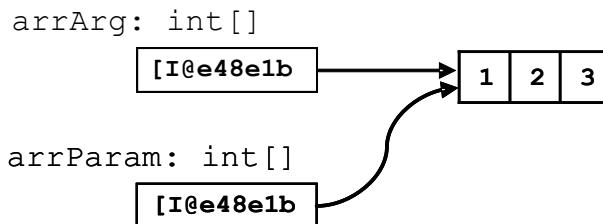
Let's also declare a method **Main()**, from which we invoke the newly created method **ModifyArray()**:

```
static void Main()
{
  int[] arrArg = new int[] { 1, 2, 3 };

  Console.Write("Before ModifyArray() the argument is: ");
  PrintArray(arrArg);

  // Modifying the array's argument
  ModifyArray(arrArg);

  Console.Write("After ModifyArray() the argument is: ");
  PrintArray(arrArg);
}
```

What would be the result of the code execution? Let's take a look:

```
Before ModifyArray() the argument is: [1, 2, 3]
In ModifyArray() the param is: [5, 2, 3]
After ModifyArray() the argument is: [5, 2, 3]
```

It is apparent that after execution of the method **ModifyArray()**, the array to which the variable **arrArg** refer, does not consists of **[1,2,3]**, but **[5,2,3]** instead. What does this mean?

The reason for such result is the fact that by passing arguments of reference type, only the value of the variable that keeps the address to the object is copied. Note that this **does not copy the object itself**.

By passing the argument that are of reference type, the only thing that is copied is the variable that keeps the reference to the object, but not the object data.

Let's try to illustrate what just was explained. We will use few drawings for the example we used above. By invocation of the method **ModifyArray()**, the value of the parameter **arrParam** is not defined and it does not keep a reference to any particular object (not a real array):

By the time of **ModifyArray()** invocation, the value that is kept in the argument **arrArg** is copied to the parameter **arrParam**:

This way, copying the reference to the elements of the array in the memory from the argument into the parameter, we tell the parameter to point to the same object, to which the argument points:

This actually is where we have to be very careful. If **the invoked method modifies the object**, to which a reference is passed, this may affect the execution of the code after the method invocation (as we have seen in the example – the method **PrintArray()** does not print the array, that was initially passed).

The difference between dealing with arguments of primitive and reference type is in the way they are passed: **primitive types are passed by their values**, **the objects, however, are passed by reference**.

Passing of Expressions as Method Arguments

When a method is invoked, we can pass a **whole expression** instead of arguments. By doing so, C# calculates the values for those expressions and by the time of code execution (if it is possible this is done at compile time) replaces the expression with its result, when the method is invoked. The following code shows methods invocation, by passing **expressions as method arguments**:

```
PrintSign(2 + 3);

float oldQuantity = 3;
float quantity = 2;
PrintMax(oldQuantity * 5, quantity * 2);
```

The result of those methods execution is:

```
Positive
Maximal number: 15.0
```

When a method with parameters is invoked, we must be aware of some specific rules, which will be explained in the next few subsections.

Passing of Arguments Compatible with the Parameter Type

We must know that we can pass only arguments that are of type **compatible** with the related parameter, declared in the method's parameters list.

For example, if the parameter that the method expects in its declaration is of type **float**, by invocation of the method we can pass a value that is of type **int**. It will be converted by the compiler to a value of type **float** and then will be passed to the method for its execution:

```
static void PrintNumber(float number)
{
  Console.WriteLine("The float number is: {0}", number);
}

static void Main()
{
  PrintNumber(5);
}
```

In the example, by invocation of **PrintNumber()** in the method **Main()**, first the integer literal **5** (that implicitly is of type **int**) is converted to the related floating point value **5.0f**. Then the so converted value is passed to the method **PrintNumber()**.

As can be expected, the result of that code execution is:

```
The float number is: 5.0
```

Compatibility of the Method Parameter and the Passed Value

The result from the calculation of an expression, passed as argument, must be **of the same type**, as the type of the declared parameter is, or **compatible** with that type (refer to the passage above).

So if a parameter of type **float** is required, we can pass the value calculated by an expression that is of a type **int**. E.g. in the example above, if instead of **PrintNumber(5)**, we called the method, with **5** replaced by the expression **2+3**, the result of the calculation of that expression must be of type **float** (one that the method expects), or of a type that can be converted to **float** with no loss (in our case this is **int**). So let's modify a little the method **Main()** from the passage above, to illustrate what just was explained:

```
static void Main()
{
  PrintNumber(2 + 3);
}
```

In this example first the summing will be executed. Then the integer result **5** will be converted to its floating point equivalent **5.0f**. When this is done the method **PrintNumber(…)** will be invoked with argument **5.0f**. The result again will be:

```
The float number is: 5.0
```

Keeping the Declaration Sequence of the Arguments Types

Values, that are passed to the method, in the time of its invocation, must be in the **same order** as the parameters are declared in the parameters list. This is due to the method signature, mentioned above.

To clarify, let's discuss the following example: we have a method **PrintNameAndAge()**, in which method declaration is a parameters list, with parameters of type's **string** and **int**, ordered as shown below**:**

Person.cs

```
class Person
{
  static void PrintNameAndAge(string name, int age)
  {
    Console.WriteLine("I am {0}, {1} year(s) old.", name, age);
  }
}
```

Let's add a method **Main()** to our class, in that method we will invoke the **PrintNameAndAge()** method. Now let's try to pass parameters in reverse (as types) order, so instead **"John"** and **25**, we will use **25** and **"John"**:

```
static void Main()
{
  // Wrong sequence of arguments
  Person.PrintNameAndAge(25, "John");
}
```

The compiler in this case will not be able to find a method that is called **PrintNameAndAge**, which accepts parameters in the sequence **int** and **string**. That is why, the compiler will notify for an error:

```
The best overloaded method match for
'Person.PrintNameAndAge(string, int)' has some invalid arguments
```

Variable Number of Arguments (var-args)

So far, we examined declaration of methods for which the parameters list coincides with the count of the arguments we pass to that method, by its invocation.

Now we will see how to declare methods that allow the count of arguments to be different any time the method is invoked, so to meet the needs of the invoking code. Such methods are often called methods with a **variable number of arguments**.

Let's we look at the example, that calculates the sum of a given array of book prices, the one that already was explained above. In that example, as a parameter we passed an array of type **decimal** that consists of the prices of the chosen books:

```
static void PrintTotalAmountForBooks(decimal[] prices)
{
  decimal totalAmount = 0;

  foreach (decimal singleBookPrice in prices)
  {
    totalAmount += singleBookPrice;
  }
  Console.WriteLine(
    "The total amount of all books is:" + totalAmount);
}
```

Defined in this way, the method suppose, that always before its invocation, we will have created an array with numbers of type **decimal** and they will be initialized with certain values.

After we created a C# method that accepts variable number of parameters, is possible, whenever a list of parameters **from the same type** must be passed, instead of passing the array that consists of those values, to pass them directly, as arguments, separated by comma.

In our case with the books, we need to create a new array, especially for that method invocation:

```
decimal[] prices = new decimal[] { 3m, 2.5m };
PrintTotalAmountForBooks(prices);
```

However, if we add some code (we will see it in a moment) to the method declaration, we will be able to directly pass list with the books prices, as method arguments:

```
PrintTotalAmountForBooks(3m, 2.5m);
PrintTotalAmountForBooks(3m, 5.1m, 10m, 4.5m);
```

Such invocation is possible only if we have declared the method in a way, so it accepts variable number of arguments (var-args).

How to Declare Method with Variable Number of Arguments

Formally the declaration of a method with variable number of arguments is the same as the declaration of any other method:

```
static <return_type> <method_name>(<parameters_list>)
{
  // Method's body
}
```

The difference is that the **<parameters_list>** is declared with the keyword **params** in the way shown below:

```
<parameters_list> =
  [<type1> <name1>[, <typei> <namei>], params <var_type>[]
<var_name>]
where i= 2, 3, …
```

The last element from the list declaration – **<params>**, is the one that allows passing of random count of arguments of type **<var_type>**, for each invocation of the method.

In the declaration of that element, before its type **<var_type>** we must add **params**: "**params <var_type>[]**". The type **<var_type>** can be either primitive or by reference.

Rules and special characteristics for the other elements from the method's parameters list, that precede the var-args parameter **<var_name>**, are the same, as those we discussed in the section "Method Parameters".

To clarify what was explained so far, we will discuss an example for declaration and invocation of a method with variable number if arguments:

```
static long CalcSum(params int[] elements)
{
  long sum = 0;
  foreach (int element in elements)
  {
```

```
        sum += element;
    }
    return sum;
}

static void Main()
{
    long sum = CalcSum(2, 5);
    Console.WriteLine(sum);

    long sum2 = CalcSum(4, 0, -2, 12);
    Console.WriteLine(sum2);

    long sum3 = CalcSum();
    Console.WriteLine(sum3);
}
```

The example sums the numbers, as their count is not known in advance. The method can be invoked with one, two or more parameters, as well as with no parameters at all. If we execute the example we will get the following result:

```
7
14
0
```

Variable Number of Arguments: Arrays vs. "params"

From the formal definition, given above, of parameter that allows passing of **variable number of arguments** by the method invocation – **<var_name>**, is actually a name of an **array of type <var_type>**. By the method invocation, the arguments of type **<var_type>** or compatible type that we pass to the method (with no care for their count) will be kept into this array. Then they will be used in the method body. The access and dealing with these parameters is in the same way we do when we work with arrays.

To make it clearer we will modify the method that calculates the sum of the prices of chosen books, to get variable number of arguments:

```
static void PrintTotalAmountForBooks(params decimal[] prices)
{
    decimal totalAmount = 0;

    foreach (decimal singleBookPrice in prices)
    {
        totalAmount += singleBookPrice;
    }
```

```
    Console.WriteLine("The total amount of all books is:" +
        totalAmount);
}
```

As we can see the only change is to change the declaration of the array **prices** with adding **params** before **decimal[]**. In the body of our method, "**prices**" is still an array of type **decimal**, so we use it in the same way as before.

Now we can invoke our method, with no need to declare in advance an array of number and pass it as an argument:

```
static void Main()
{
    PrintTotalAmountForBooks(3m, 2.5m);
    PrintTotalAmountForBooks(1m, 2m, 3.5m, 7.5m);
}
```

The result of the two invocations will be:

```
The total amount of all books is: 5.5
The total amount of all books is: 14.0
```

Since **prices** is an array, it can be assumed that we can declare and initialize an array before invocation of our method. Then to pass that array as an argument:

```
static void Main()
{
    decimal[] pricesArr = new decimal[] { 3m, 2.5m };

    // Passing initialized array as var-arg:
    PrintTotalAmountForBooks(pricesArr);
}
```

The above is legal invocation, and the result from that code execution is the following:

```
The total amount of all books is: 5.5
```

Position and Declaration of a Method with Variable Arguments

A method, that has a variable number of its arguments, can also have **other parameters** in its parameters list.

The following code, for example, has as a first parameter an element of type **string**, and right after it there can be one or more parameters of type **int**:

```
static void DoSomething(string strParam, params int[] x)
{
}
```

The one thing that we must consider is that the element from the parameters list in the method's definition, that allows passing of a variable number of arguments, must **always be placed at the end** of the parameters list.

	The element of the parameters list, that allows passing of variable number of arguments by invocation of a method, must always be declared at the end of the method's parameters list.

So, if we try to put the declaration of the var-args parameter **x**, shown in the last example, not at the last place, like so:

```
static void DoSomething(params int[] x, string strParam)
{
}
```

The compiler will return the following error message:

```
A parameter array must be the last parameter in a formal
parameter list
```

Limitations on the Count for the Variable Arguments

Another limitation, for the methods with variable number of arguments, is that the method cannot have in its declaration more than one parameter that allows passing of variable numbers of arguments. So if we try to compile a method declared in the following way:

```
static void DoSomething(params int[] x, params string[] z)
{
}
```

The compiler will return the already known error message:

```
A parameter array must be the last parameter in a formal
parameter list
```

This rule can be taken as a special case of the rule for the var-args position, i.e. the related parameter to be at the end of the parameters list.

Specifics of Empty Parameter List

After we got familiar with the declaration and invocation of methods with variable number of arguments, one more question arises. What would happen if we invoke such method, but with no parameters?

For example, what would be the result of the invocation of our method that calculates the sum of books prices, in a case we did not liked any book:

```
static void Main()
{
  PrintTotalAmountForBooks();
}
```

As can be seen this code is compiled with no errors and after its execution the result is as follow:

```
The total amount of all books is: 0
```

This happens because, although, we did not pass any value to our method, by its invocation, the array **`decimal[] prices`** is created, but it is empty (i.e. it does not consists of any elements).

This has to be remembered, because even if we did not initialize the array, C# takes care to do so for the array that has to keep the parameters.

Method with Variable Number of Arguments – Example

Bearing in mind how we define methods with variable number of arguments, we can write the **`Main()`** method of a C# program in the following way:

```
static void Main(params string[] args)
{
  // Method body comes here
}
```

The definition above is valid and is accepted without any errors by the compiler.

Optional Parameters and Named Arguments

Named arguments and optional parameters are two different functionalities of the C# language. However, they often are used together. These parameters are introduced in C#, version 4.0. **Optional parameters** allow some parameters to be skipped when a method is invoked. **Named arguments** on their side, allow method parameter values to be set by their name, instead of their exact position in the parameters list. These two features in the C# language syntax are very useful in cases, when we invoke a method with a different combination of its parameters.

Declaration of optional parameters can be done just by using a **default value** in the way shown below:

```
static void SomeMethod(int x, int y = 5, int z = 7)
{
}
```

In the example above **y** and **z** are optional and can be skipped upon method's invocation:

```
static void Main()
{
   // Normal call of SomeMethod
   SomeMethod(1, 2, 3);

   // Omitting z - equivalent to SomeMethod(1, 2, 7)
   SomeMethod(1, 2);

   // Omitting both y and z – equivalent to SomeMethod(1, 5, 7)
   SomeMethod(1);
}
```

We can pass a value by a particular **parameter name**, by setting the parameter's name, followed by a colon and the value of the parameter. An example of using **named arguments** is shown below:

```
static void Main()
{
   // Passing z by name and x by position
   SomeMethod(1, z: 3);

   // Passing both x and z by name
   SomeMethod(x: 1, z: 3);

   // Reversing the order of the arguments passed by name
   SomeMethod(z: 3, x: 1);
}
```

All invocations in the sample above are equivalent to each other – parameter **y is** skipped, but **x** and **z** are set to 1 and 3. The only difference between the second and third call is that the parameter values are calculated in the same order they are passed to the method, in the last invocation 3 will be calculated before 1. In this example all parameters are constants and their purpose is only to clarify the idea of **named and optional parameters**. However, the mentioned consideration may lead to some unexpected behavior when the order of parameters calculation matters.

Method Overloading

When in a class a method is declared and its name coincides with the name of another method, but their signatures differ by their **parameters list** (count of the method's parameters or the way they are arranged), we say that there are different **variations / overloads of that method** (**method overloading**).

As an example, let's assume that we have to write a program that draws letters and digits to the screen. We also can assume that our program has methods for drawing strings **DrawString(string str)**, integers – **DrawInt(int number)**, and floating point digits – **DrawFloat(float number)** and so on:

```
static void DrawString(string str)
{
  // Draw string
}

static void DrawInt(int number)
{
  // Draw integer
}

static void DrawFloat(float number)
{
  // Draw float number
}
```

As we can see the C# language allows us to create variations of the same method **Draw(…)**, called **overloads**. The method below gets combinations of different parameters, depending of what we want to write on the screen:

```
static void Draw(string str)
{
  // Draw string
}

static void Draw(int number)
{
  // Draw integer
}

static void Draw(float number)
{
  // Draw float number
```

```
}
```

The definitions of the methods above are valid and will compile without error messages. The method **Draw(…)** is also called **overloaded**.

Method Parameters and Method Signature

As mentioned above, there are only two things required in C# to specify a method signature: the **parameter type** and **the order in which the parameters are listed**. The names of the method's parameters are not significant for the method's declaration.

	The most important aspect of creating an unambiguous declaration of a method in C# is the definition of its signature and the type of the method's parameters in particular.

For example in C#, the following two declarations are actually declarations of one and the same method. That's because the parameter type in each of their parameters is the same – **int** and **float**. So the names of the variables we are using – **param1** and **param2** or **p1** and **p2**, are not significant:

```
// These two lines will cause an error
static void DoSomething(int param1, float param2) { }
static void DoSomething(int p1, float p2) { }
```

If we declare two or more methods in one class, in the way shown above, the compiler will show an **error message**, which will look something like the one below:

```
Type '<the_name_of_your_class>' already defines a member called
'DoSomething' with the same parameter types.
```

If we change the parameter type from a **given position of the parameter list to a different type**, in C# they will count as two absolutely different methods, or more precisely said, **different variations of a method with the same name**.

For example if in the second method, the second parameter from the parameter list of any of the methods – **float p2**, is declared not as **float**, but as **int for example**, we will have two different methods with two different signatures – **DoSomething(int, float)** and **DoSomething(int, int)**. Now the second element from their signature – **parameter list**, is different, due to difference of their second element type:

```
static void DoSomething(int p1, float p2) { }
static void DoSomething(int param1, int param2) { }
```

In this case even if we type the same name for the parameters, the compiler will accept this declaration, because they are practically different methods:

```
static void DoSomething(int param1, float param2) { }
static void DoSomething(int param1, int param2) { }
```

The compiler will accept the code again if we declare two variations of the method, but this time we are going to change the order of the parameters instead of their type.

```
static void DoSomething(int param1, float param2) { }
static void DoSomething(float param2, int param1) { }
```

In the example above the **order of the parameter types** is different and this makes the signature different too. Since the parameter lists are different, it plays no role that the name (**DoSomething**) is the same for both methods. We still have different signatures for both methods.

Overloaded Methods Invocation

Since we have declared methods with the same name and different signatures, we can invoke each of them as any other method – just by using their name and arguments. Here is an example:

```
static void PrintNumbers(int intValue, float floatValue)
{
  Console.WriteLine(intValue + "; " + floatValue);
}

static void PrintNumbers(float floatValue, int intValue)
{
  Console.WriteLine(floatValue + "; " + intValue);
}

static void Main()
{
  PrintNumbers(2.71f, 2);
  PrintNumbers(5, 3.14159f);
}
```

When the code executes, we will see, that the first invocation refers to the second method, and the second invocation refers to the first method. Which method will be invoked depends on the type of the used parameters. The result after executing the code above is:

```
2.71; 2
5; 3.14159
```

The lines below, however, **will not compile** and execute:

```
static void Main()
{
  PrintNumbers(2, 3);
}
```

The reason for this not to work is that the compiler tries to convert both integer numbers to suitable types before passing them to any of the methods named **PrintNumbers**. In this case, however, these conversions are not equal. There are two possible options – either to convert the first parameter to **float** and call the method **PrintNumbers(float, int)** or to convert the second parameter to **float** and call the method **PrintNumbers(int, float)**. This ambiguity has to be manually resolved, and one way to do so is shown in the example below:

```
static void Main()
{
  PrintNumbers((float)2, (short)3);
}
```

The code above will be compiled without errors, because after the arguments are transformed, it is clearly decided which method we refer to – **PrintNumbers(float, int)**.

Methods with Coinciding Signatures

We will discuss some other interesting examples that show how to use methods. Let's take a look at an example of an incorrect redefinition (overload) of methods:

```
static int Sum(int a, int b)
{
  return a + b;
}

static long Sum(int a, int b)
{
  return a + b;
}

static void Main()
{
  Console.WriteLine(Sum(2, 3));
}
```

The code from the example will show an **error message** upon compilation process, because there are two methods with same parameters lists (i.e. with same signature) which return results of different types. This makes the method invocation ambiguous, so it is not allowed by the compiler.

Triangles with Different Size – Example

It would be a good time now to give a little bit more complex example, since we know now how to declare methods with parameters, how to invoke them as well as how to get result back from those methods. Let's assume we want to write a program, which prints **triangles** on the console, as those shown below:

```
n = 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

n = 6
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

A possible solution of this task is given below:

Triangle.cs

```
using System;

class Triangle
{
```

```
    static void Main()
    {
        // Entering the value of the variable n
        Console.Write("n = ");
        int n = int.Parse(Console.ReadLine());
        Console.WriteLine();

        // Printing the upper part of the triangle
        for (int line = 1; line <= n; line++)
        {
            PrintLine(1, line);
        }

        // Printing the bottom part of the triangle
        // that is under the longest line
        for (int line = n - 1; line >= 1; line--)
        {
            PrintLine(1, line);
        }
    }

    static void PrintLine(int start, int end)
    {
        for (int i = start; i <= end; i++)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();
    }
}
```

Let's discuss how the example code works. We should think of the triangles as sequences of numbers, placed on separate lines, since we can print each line directly on the console. In order to print each line of the triangle on the console we need a tool. For this purpose we created the method **PrintLine(…)**.

In this method, by using a **for**-loop, we print a line of consequent numbers. The first number from this sequence is the first parameter from the method's parameter list (the variable **start**). The last element of the sequence is the number, passed to the method, as second parameter (the variable **end**).

Notice that since the numbers are sequential, the length (count of the numbers) of each line corresponds to the difference between the second parameter **end** and the first one – **start**, from the methods parameters list (this will be useful later, when we build the triangles).

Then we implement an algorithm that prints the triangles, as whole figures, in the **Main() method**. With another method **int.Parse**, we get the **n** variable and print the empty line.

Now with two sequential **for**-loops we build the triangle according to the entered **n**. With the first loop we print all the lines that draw the upper part of the triangle and the middle (longest) line inclusively. With the second loop, we print the rest of the triangle's lines that lie below the middle line.

As we mentioned above, the line number, corresponds to the element count placed on the appropriate line. And since we always start from 1, the line number will always be equal to the last element in the sequence, which has to be printed on that line. So, we can use this when we call **PrintLine(…)**, as it requires exactly that for its parameters.

Note that, the count of the elements on each next line, increases with one and so the last element of each sequent line must be greater (one is added) than the last element of the preceding line. That's why at each loop iteration of the first for-loop, we pass to the **PrintLine(…) method**, as first parameter **1**, and as a second – the current value of the variable **line**. Since, on each execution of the body of the loop, **line** increases with one, at each iteration **PrintLine(…)** the method prints a line that has more than one element than the preceding line.

With the second loop, that draws the part under the middle triangle line, we follow the reverse logic. The downward we print lines, the shorter lines we print. Each line decreases with one element according to its preceding line. Hence, we set the initial value for the variable **line** in the second loop: **line = n-1**. After each iteration of the loop **line** decreases with one and pass it as second parameter to the **PrintLine(…)**.

We can improve the program, as we take the logic that prints the triangle, in a separate method. It can be noticed that, logically, the triangle print is clearly defined, that is why we can declare a method with one parameter (the value that we get from the keyboard) and to invoke it from the **Main()** method:

```
static void Main()
{
    Console.Write("n = ");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine();

    PrintTriangle(n);
}

static void PrintTriangle(int n)
{
    // Printing the upper part of the triangle
```

```
    for (int line = 1; line <= n; line++)
    {
        PrintLine(1, line);
    }

    // Printing the bottom part of the triangle
    // that is under the longest line
    for (int line = n - 1; line >= 1; line--)
    {
        PrintLine(1, line);
    }
}
```

If we execute the program and enter for **n** the value 3, we will get the following result:

```
n = 3

1
1 2
1 2 3
1 2
1
```

Returning a Result from a Method

So far, we always were given examples, in which the method does something like printing on the console, and nothing more. Methods, however, usually do not just execute a simple code sequence, but in addition they often **return results**. So let's take a look at how this actually happens.

Declaring a Method that Returns a Result

Let's see again how to declare a method.

```
static <return_type> <method_name>(<parameters_list>)
```

Earlier we said that at the place of **<return type>** we will always put **void**. Now we will extend this definition, as we will see, that **void** is not the only choice. Instead of void we can return any type either primitive (**int**, **float**, **double**, …) or by reference (as **string** or array), depending on the type of the result that the method shall return after its execution.

For example, take a method that calculates the area of a square and instead of printing it to the console **returns it as a result**. So, the declaration would look as follows:

```
static double CalcSquareSurface(double sideLength)
```

As can be seen the result of the calculation of the area is of type **double**.

How to Use the Returned Value?

When the method is executed and returns a value, we can imagine that C# puts this value where this method has been invoked from. Then the program continues work with that value. Respectively, that returned value, we can use for any purpose from the calling method.

Assigning to a Variable

We can also assign the result of the method execution to a variable of an appropriate type:

```
// GetCompanyLogo() returns a string
string companyLogo = GetCompanyLogo();
```

Usage in Expressions

After a method returns a **result**, it can be used then in **expressions** too.

So for example, to find the total price for invoice calculation, we must get the single price and to multiply it by the quantity:

```
float totalPrice = GetSinglePrice() * quantity;
```

Using the Returned Value as Method Parameter

We can pass the result from the method execution as value in the parameters list from another method:

```
Console.WriteLine(GetCompanyLogo());
```

In this example, in the beginning we invoke the method **GetCompanyLogo()**, and write it as an argument of the method **WriteLine()**. Right after the **GetCompanyLogo()** method finishes its execution it will return a result. Let's say that the result will be "**Microsoft Corporation**". Then C# will put the result returned by the method's execution in the method's place. So we can assume that this is represented in the code in the following way:

```
Console.WriteLine("Microsoft Corporation");
```

Returned Value Type

As it was already explained above, the result that a method returns can be of any type – **int**, **string**, array and so on. When, however, instead of a type

we use the keyword **void** instead of a type, this mean that method does not return value.

The Operator "return"

To make a method return value, the keyword **return** must be placed in the method’s body, followed by an **expression** that will be returned as a result by the method:

```
static <return_type> <method_name>(<parameters_list>)
{
  // Some code that is preparing the method's result comes here
  return <method's_result>;
}
```

Respectively **<method's_result>**, is of type **<return_type>**. For example:

```
static long Multiply(int number1, int number2)
{
  long result = number1 * number2;
  return result;
}
```

In this method after the multiplication, by using the **return** the method will produce as a result of its execution the integer variable **result**.

Compatibility of the Result and the Retuning Type

The result returned by the method, can be of a type that is **compatible** (the one that can be implicitly converted) with the type of the returned value **<return_type>**.

For example, we can modify the following example, in which the type of the returned value to be of type **float**, but not **int** and to keep the following code in the shown way:

```
static float Multiply(int number1, int number2)
{
  int result = number1 * number2;
  return result;
}
```

In this case after the multiplication execution, the result will be of type **int**. Even though the type of the expression after the **return** keyword is not of type **float**, it can be returned, because it can be implicitly converted to **float**.

Using an Expression after the Return Operator

It is allowed (whenever this will not make the code look complicated / ugly) to directly put some expression after the keyword **return**:

```
static int Multiply(int number1, int number2)
{
  return number1 * number2;
}
```

In this situation, after the calculation of **number1 * number2**, the result that this expression produces will be replaced where the expression is, and hence will be returned by the **return** operator.

Features of the Return Operator

The execution of **return** does two things:

- **Stops immediately** the method execution.
- **Returns the result** of the executed method to the calling method.

In relation to the first feature of **return operator**, we must note that, since it stops the method's execution (and no code after it and before the method body's closing bracket will be executed), we should not put any code after the **return** operation.

Though, if we do so, the compiler will show a **warning message**:

```
static int Add(int number1, int number2)
{
  int result = number1 + number2;
  return result;

  // Let's try to "clean" the result variable here:
  result = 0;
}
```

In this example the compilation will be successful, but for the lines after **return**, the compiler will output a warning message like this:

```
Unreachable code detected
```

When the method has **void** for returned value type, then after **return**, there would be no expression to be returned. In that case **return** usage is only used to **stop the method's execution**:

```
static void PrintPositiveNumber(int number)
{
```

```
    if (number <= 0)
    {
        // If the number is NOT positive, terminate the method
        return;
    }
    Console.WriteLine(number);
}
```

Multiple Return Statements

The last thing that must be said about the operator **return** is that it can be called from several places in the code of our method, but should be guaranteed that at least one of the operators **return** that we have used, will be reached while executing the method.

So let's take a look, at the example for a method that gets two numbers, and then upon their values return 1 if the first is greater than the second, 0 if both are equal, or -1 if the second is greater than the first:

```
static int CompareTo(int number1, int number2)
{
    if (number1 > number2)
    {
        return 1;
    }
    else if (number1 == number2)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
```

Having **multiple return statements** is usual in programming and is typical for methods that check several cases, like the above.

Why Is the Returned Value Type not a Part of the Method Signature?

In C# it is not allowed to have several methods that have equal name and parameters, but different type of returned value. This means that the following code **will fail to compile**:

```
static int Add(int number1, int number2)
```

```
{
  return (number1 + number2);
}

static double Add(int number1, int number2)
{
  return (number1 + number2);
}
```

The reason for this limitation is that the compiler doesn't know which of both methods must be invoked. Both methods have the **same signature** (sequence of parameters along with their types). Note that the return value is not part of the method's signature. That is why on the declaration of the methods an error message like the one below will be returned:

```
Type '<the_name_of_your_class>' already defines a member called
'Add' with the same parameter types
```

Where **`<the_name_of_your_class>`** is the name of the class in which we have tried to declare those methods.

Fahrenheit to Celsius Conversion – Example

Now we have to write a program that for a given (by the user) body temperature, measured in Fahrenheit degrees, has to convert that temperature and output it in Celsius degrees, with the following message: "**Your body temperature in Celsius degrees is X**", where **X** is respectively the Celsius degrees. In addition if the measured temperature in Celsius is higher than 37 degrees, the program should warn the user that they are ill, with the following message "**You are ill!**".

For starters, we can make fast **research in Internet** and find out that the **Celsius to Fahrenheit formula** is like this one: **`°C = (°F - 32) * 5 / 9`**, where respectively with **`°C`** we mark the temperature measured in Celsius, and with **`°F`** – the temperature in Fahrenheit.

After analysis of the current task, we can see that it can be divided to subtasks as follow:

- Take the temperature measured in Fahrenheit degrees as an **input** from the console (the user must enter it).
- **Convert** that number to its corresponding value, for temperature measured in Celsius.
- **Print** a message for the converted temperature in Celsius.
- If the temperature is found to be **higher than 37 °C**, print a message that the user is ill.

A sample implementation of the above described algorithm is given below in the class **TemperatureConverter**:

TemperatureConverter.cs

```
using System;

class TemperatureConverter
{
  static double ConvertFahrenheitToCelsius(double temperatureF)
  {
    double temperatureC = (temperatureF - 32) * 5 / 9;
    return temperatureC;
  }

  static void Main()
  {
    Console.Write(
      "Enter your body temperature in Fahrenheit degrees: ");
    double temperature = double.Parse(Console.ReadLine());

    temperature = ConvertFahrenheitToCelsius(temperature);

    Console.WriteLine(
      "Your body temperature in Celsius degrees is {0}.",
      temperature);

    if (temperature >= 37)
    {
      Console.WriteLine("You are ill!");
    }
  }
}
```

The operations for input of the temperature and output of the messages are **trivial**, so we will skip their explanation, as we will focus on the approach to convert the temperatures. As we can see this is a logical unit that can be separated in its own method. By doing so, not only the program source code will get clearer, but moreover, we will have the opportunity to reuse that piece of code, whenever we need it, so we just will use the same method. So we declare the method **ConvertFahrenheitToCelsius(…)**, with list of one parameter with the name **temperatureF** that represents the measured value of the temperature in Fahrenheit. Then the method returns a result of type **double**, which represents the calculated body temperature in Celsius degrees. In the method’s body we use the formula we found on Internet (and write it according to the C# syntax).

Since we are done with this step from our task solution, we have decided that the rest of the steps we will not need to be in separate methods, so we just implement them in the **Main()** method of the class.

By the method **double.Parse(…)**, we get the user's body temperature as we have previously asked him for it, by the following message: "**Enter your body temperature in Fahrenheit degrees**".

Then we invoke the method **ConvertFahrenheitToCelsius()** and we store the returned result in the variable **temperature**.

By the method **Console.WriteLine()** we print the message "**Your body temperature in Celsius degrees is X**", where **X** is replaced with the value of **temperature**.

The last step we must make is to check whether the temperature is higher than 37 degrees in Celsius or no. This can be done by using a conditional statement **if**. So if the temperature is higher than 37 degrees Celsius a message that the user is ill must be printed.

Below is shown a possible output of the program:

```
Enter your body temperature in Fahrenheit degrees: 100
Your body temperature in Celsius degrees is 37,777778.
You are ill!
```

Difference between Two Months – Example

Let's take a look at the following task: we have to write a program which, by given two numbers, that are between **1** and **12** (so to correspond to a particular month) prints the count of months between these months. The message that must be printed to the console must be "**There is X months period from Y to Z.**", where **X** is the count of the months, that we must calculate, and **Y** and **Z**, are respectively the names of the months that mark start and end of the period.

By reading carefully the task we will try to divide it into subtasks, that can be more easily solved, and then by combining them to get the whole solution. We can see that we have to solve the following subtasks:

- To **enter** the months numbers that mark beginning and end of the period.
- To **calculate** the period between the input months.
- To **print** the message.
- In the message instead of the numbers we entered, for beginning and end of the period, we must write their corresponding **month names** in English.

A possible solution of the given task is shown below:

Months.cs

```
using System;

class Months
{
  static string GetMonthName(int month)
  {
    string monthName;
    switch (month)
    {
      case 1:
         monthName = "January";
         break;
      case 2:
        monthName = "February";
        break;
      case 3:
        monthName = "March";
        break;
      case 4:
        monthName = "April";
        break;
      case 5:
        monthName = "May";
        break;
      case 6:
        monthName = "June";
        break;
      case 7:
        monthName = "July";
        break;
      case 8:
        monthName = "August";
        break;
      case 9:
        monthName = "September";
        break;
      case 10:
        monthName = "October";
        break;
      case 11:
        monthName = "November";
        break;
```

```
            case 12:
                monthName = "December";
                break;
            default:
                Console.WriteLine("Invalid month!");
                return null;
        }
        return monthName;
    }

    static void SayPeriod(int startMonth, int endMonth)
    {
        int period = endMonth - startMonth;
        if (period < 0)
        {
            // Fix negative distance
            period = period + 12;
        }
        Console.WriteLine(
            "There is {0} months period from {1} to {2}.",
            period, GetMonthName(startMonth),
            GetMonthName(endMonth));
    }

    static void Main()
    {
        Console.Write("First month (1-12): ");
        int firstMonth = int.Parse(Console.ReadLine());

        Console.Write("Second month (1-12): ");
        int secondMonth = int.Parse(Console.ReadLine());

        SayPeriod(firstMonth, secondMonth);
    }
}
```

The first task solution is trivial. In the **Main()** method we will use **int.Parse(…)** so we get the months for the period, the length of which we aim to calculate.

Then we see that period calculation and message printing can be logically separated as a subtask, so we create a method **SayPeriod(…)** that has two parameters – numbers representing month numbers that mark the beginning and the end of the period. This method will not return a value but it will calculate period and print the message, described in the task, to the console, by the standard output – **Console.WriteLine(…)**.

Apparently, to find the length of the period between two months, we have to subtract the number of the beginning month from that of the end month. We consider also, that if the second month has number less that the number of the first month, then the user most probably has had the assumption that the second month is not in the current year, but in the next one. That is why, if the difference between the two months is negative, we must add **12** to it – the length of a year in months, and so to find the length of the given period. Then we must print the message, as for the months names we use the method **GetMonthName(…)**.

The method that gets the month's name by its number can be easily created with conditional **switch-case** statement, in which we could get the months for each of the input numbers. If the value is not in the range of [1…12], the program will **report an error**. Later in the chapter "Exception Handling" we will discuss in details how to notify for an error occurring. You will be shown how to catch and deal with the exceptions (error notifications). However, for now we just will print an error message to the console. This is generally an incorrect behavior and we will learn how to avoid it in the chapter "High-Quality Code", section "What Should a Method Do".

At the end, in the **Main()** method we invoke the **SayPeriod()** method, by entered numbers for beginning and end of the period. By doing so, we have completely solved the task.

A possible output, if the input was 2 and 6, is shown below:

```
First month (1-12): 2
Second month (1-12): 6
There is 4 months period from February to June.
```

Input Data Validation – Example

In this task we must write a program that asks the user what time it is, by printing on the console "**What time is it?**". Then the user must enter two numbers – one for hours and one for minutes. If the input data represents a valid time, the program must output the message "**The time is hh:mm now.**", where **hh** respectively means the hours, and **mm** – the minutes. If the entered hours or minutes are not valid, the program must print the message "**Incorrect time!**".

After we read the task carefully, we see that it can be divided into the following subtasks:

- Get input data for hours and minutes.
- Check if input data is valid (input validation).
- Print the corresponding message – either an error message, or the valid time message.

We consider that getting the input data and printing the output messages will not be a problem anymore, so we will focus on input data validation, i.e. validation the numbers for hours and minutes. We know that the hours are in the range from 0 to 23 inclusive, and the minutes respectively from 0 to 59 inclusive. Since the data (for hours and for minutes) has not the same nature, we decide to create two separate methods. One of them will check the validity of hours, while the other will check the validity for minutes.

A solution is shown below:

DataValidation.cs

```
using System;

class DataValidation
{
  static void Main()
  {
    Console.WriteLine("What time is it?");

    Console.Write("Hours: ");
    int hours = int.Parse(Console.ReadLine());

    Console.Write("Minutes: ");
    int minutes = int.Parse(Console.ReadLine());

    bool isValidTime =
      ValidateHours(hours) && ValidateMinutes(minutes);
    if (isValidTime)
    {
      Console.WriteLine("The time is {0}:{1} now.",
        hours, minutes);
    }
    else
    {
      Console.WriteLine("Incorrect time!");
    }
  }

  static bool ValidateHours(int hours)
  {
    bool result = (hours >= 0) && (hours < 24);
    return result;
  }

  static bool ValidateMinutes(int minutes)
```

```
  {
    bool result = (minutes >= 0) && (minutes <= 59);
    return result;
  }
}
```

The method that checks the hours is named **ValidateHours()**, and it gets a number of type **int** for the hours, and returns result of type **bool**, i.e. **true** if the input number is a valid hour, otherwise – **false**:

```
static bool ValidateHours(int hours)
{
  bool result = (hours >= 0) && (hours < 24);
  return result;
}
```

We use simple logic to declare method, which checks the validity of the minutes. We named it **ValidateMinutes()**, since it gets a parameter that is integer value and represents the minutes, and returns a value of type **bool**. If the input number is a valid minute value, the method will return as result **true**, otherwise – **false**:

```
static bool ValidateMinutes(int minutes)
{
  bool result = (minutes >= 0) && (minutes <= 59);
  return result;
}
```

Since we are done with the most complicated part of the task, we declare the **Main() method**. In its body we print out the question according to the task – "**What time is it?**". Then by the method **int.Parse(…)**, we read from the console the numbers for hours and minutes, then the results are kept in the integer variables **hours** and **minutes**:

```
Console.WriteLine("What time is it?");

Console.Write("Hours: ");
int hours = int.Parse(Console.ReadLine());

Console.Write("Minutes: ");
int minutes = int.Parse(Console.ReadLine());
```

The result from the validation is kept in a variable of type **bool** – **isValidTime**, as we sequentially invoke the methods we have already declared – **ValidateHours()** and **ValidateMinutes()**, as of course we pass

the appropriate variables **hours** and **minutes** to each of them. To validate the input data as a whole, we unite the results from the methods invocation with the operator for logical "and" **&&**:

```
bool isValidTime =
  ValidateHours(hours) && ValidateMinutes(minutes);
```

After we stored the result, telling us whether the input data is valid or not, in the variable **isValidTime**, we use the conditional statement **if**, cope with the last problem for the given task – Printing the information to the user, whether the input is valid or not. With the method **Console.WriteLine(…)**, if **isValidTime** is **true**, we print on the console "**The time is hh:mm now.**" where **hh** is respectively the value of the variable **hours**, and **mm** – of the variable **minutes**. In the **else** part of the conditional statement we print that the input time was invalid – "**Incorrect time!**".

A possible output of the program, with correct data, is shown below:

```
What time is it?
Hours: 17
Minutes: 33
The time is 17:33 now.
```

And here's how the program behaves, when the data is incorrect:

```
What time is it?
Hours: 33
Minutes: -2
Incorrect time!
```

Sorting – Example

Let's try to create a method that sorts (puts in order) a set of values in ascending order. The result will be a string with the sorted numbers.

With this in mind, we suppose that the subtasks we have to cope with are two:

- How to give the numbers to our method, so it could sort them
- How to sort those numbers

Our method has to take an array on numbers as a parameter, create a sort of that array and return it:

```
static int[] Sort(int[] numbers)
{
  // The sorting logic comes here …
```

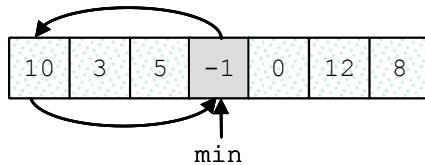
```
    return numbers;
}
```

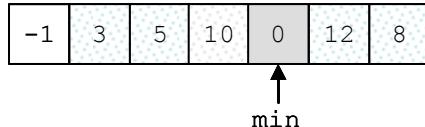
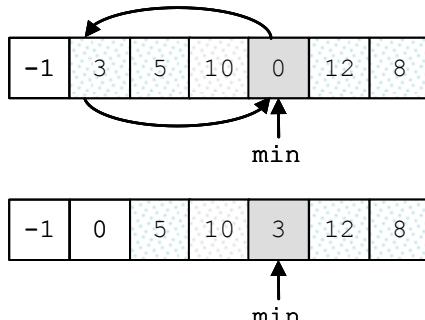
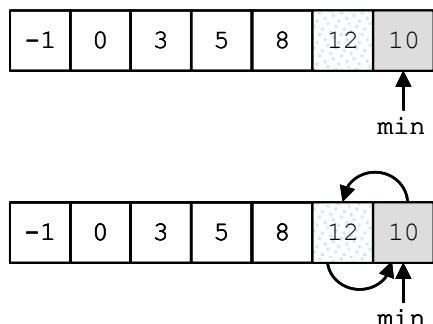
This solution seems to satisfy the task requirements. However, it seems that we could optimize it more, and instead of the argument to be an integer array, we can declare it in such way that it could accept a variable count of integer parameters.

This will save us the need to initialize the array in advance when we invoke the method with a small set of numbers. In case of bigger sets of input numbers, as we saw in the subsection for method declaration with a variable number of arguments, we could directly pass an already initialized array of integers, instead of passing them as parameters of the method. Hence, the initial declaration turns into:

```
static int[] Sort(params int[] numbers)
{
    // The sorting logic comes here …

    return numbers;
}
```

Now we must to decide how to sort our array. One of the easiest ways for this to be done is to use the so-called **"selection sort" algorithm**. This method considers the array as two parts – sorted and unsorted. The sorted part is in the left side of the array, while the unsorted is in the right. For each step of the algorithm, the sorted part expands to the right with one element and the unsorted shrinks with one element from its left part.

Let’s take a look at an example. So assume we have the following unsorted array and we want to order its elements by **selection sorting**:

10	3	5	-1	0	12	8

On each step our algorithms must find the minimal element in the unsorted part of the array:

Then the minimal element must swap with the first element from the unsorted part of the array:

Then we look for the minimal element again, from the rest of the unsorted part of the array (all elements except the first one):

That minimal element now exchanges with the first from the unsorted part:

So this step is repeated until the unsorted part of the array reaches a length of 0, i.e. it is empty:

As a result the array is **sorted**:

-1	0	3	5	8	10	12

This is a variant of a code, which implements the algorithm explained above (selection sort):

```
static int[] Sort(params int[] numbers)
{
   // The sorting logic:
```

```
    for (int i = 0; i < numbers.Length - 1; i++)
    {
        // Loop operating over the unsorted part of the array
        for (int j = i + 1; j < numbers.Length; j++)
        {
            // Swapping the values
            if (numbers[i] > numbers[j])
            {
                int temp = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = temp;
            }
        }
    } // End of the sorting logic
    return numbers;
}
```

Let's declare a method **PrintNumbers(params int[])** that outputs the list with numbers to the console, and then to test this example by writing a few lines directly into the **Main(…)** method:

SortingEngine.cs

```
using System;

class SortingEngine
{
    static int[] Sort(params int[] numbers)
    {
        // The sorting logic:
        for (int i = 0; i < numbers.Length - 1; i++)
        {
            // Loop that is operating over the un-sorted part of
            // the array
            for (int j = i + 1; j < numbers.Length; j++)
            {
                // Swapping the values
                if (numbers[i] > numbers[j])
                {
                    int oldNum = numbers[i];
                    numbers[i] = numbers[j];
                    numbers[j] = oldNum;
                }
            }
```

```
    } // End of the sorting logic
    return numbers;
  }

  static void PrintNumbers(params int[] numbers)
  {
    for (int i = 0; i < numbers.Length; i++)
    {
      Console.Write("{0}", numbers[i]);
      if (i < (numbers.Length - 1))
      {
        Console.Write(", ");
      }
    }
  }

  static void Main()
  {
    int[] numbers = Sort(10, 3, 5, -1, 0, 12, 8);
    PrintNumbers(numbers);
  }
}
```

After this code is compiled and executed, the result is exactly as the one that was expected – the array is ordered ascending:

```
-1, 0, 3, 5, 8, 10, 12
```

Best Practices when Using Methods

In the chapter "High-Quality Programming Code" we will explain in details about the good practices for writing methods. None the less, we will look at some of them right now, so we can start applying the good practices and start developing a good programming style:

- Each method must resolve a **distinct, well defined task**. This feature is also known as **strong cohesion**, i.e. to give a focus onto one single task, not to several tasks no strongly related logically. A single method should perform a single task, its code should be well structured, easy to understand, and easy to be maintained. One method must NOT solve several tasks!

- A method has to have a **good name**, i.e. name that is descriptive and from which becomes clear **what the method does**. As an example: a method that sorts numbers should be named **SortNumbers()**, but should not be named **Number()** or **Processing()** or **Method2()**. If it

cannot be given a good name, this may indicate that the method solves more than one task and, hence, it must be separated into sub-methods.

- Method names should **describe an action**, so they should contain a **verb** or a **verb + noun** (possibly with an adjective to supplement the noun). For example good method names are **FindSmallestElement()**, **Sort(int[] arr)** and **ReadInputData()**.

- It is assumed that all the method names in C# will start with capital letter. **PascalCase** rule is used, i.e. each new word that is concatenated to the end of the method name must start with capital letter. For example: **SendEmail(…)**, but not **sendEmail(…)** or **send_email(…)**.

- A method must do whatever is described with its name, or it must return an error (throws an exception). It is not correct that the methods return wrong or unusual result when it has received invalid input data. **The method resolves the task it is created for, or returns an error**. Any other behavior is incorrect. We will discuss this principle in "High-Quality Programming Code", section "What a Method Should Do".

- A method must have **minimum dependency** to the class in which the method is declared and to other methods and classes. This feature of the methods is also known as **loose coupling**. This means that the method must do its job by using the data that passed to it as parameters, but not data that can be accessed from other places. Methods should not have **side effects** (for example to change some global variable or print something on the console in the meantime).

- It is recommended that the methods **must be short**. Methods that are longer than a computer screen must be avoided. To do so, the logic implemented in the method is divided by functionality, to several smaller sub-methods. These sub-methods are then called from the original place they were cut off.

- To improve the readability of a method and the code structure, it is good idea a functionality that is well detached logically, to be placed in a separate method. For example if we have a method that calculates the volume of a dam lake, the process of calculating the volume of a parallelepiped can be defined in a separate method. Then that new method can be invoked as many times as necessary. Hence, **the sub-task is separated from the main task**. Since the dam lake can be taken as set of many different parallelepipeds, calculating the volume of each one of them is logical detached functionality.

- The last but most important rule is that **a method should either do what it name says or throw an exception**. If a method cannot perform its job (e.g. due to incorrect input), it should **throw an exception**, not return invalid or neutral result. How to throw an exception will be explained in the chapter "Exception Handling", but for now you should remember that **returning an incorrect result or having a side effect are bad practices**. If a method cannot do its job,

it should inform its caller about this by throwing appropriate exception. Methods should **never return wrong result**!

Exercises

1. Write a code that by given name prints on the console "**Hello, <name>!**" (for example: "**Hello, Peter!**").
2. Create a method **GetMax()** with two integer (**int**) parameters, that returns **maximal** of the two numbers. Write a program that reads three numbers from the console and prints the biggest of them. Use the **GetMax()** method you just created. Write a test program that validates that the methods works correctly.
3. Write a method that returns the **English name of the last digit** of a given number. Example: for **512** prints "**two**"; for **1024** → "**four**".
4. Write a method that finds **how many times certain number can be found in a given array**. Write a program to test that the method works correctly.
5. Write a method that checks whether an element, from a certain position in an array is **greater than its two neighbors**. Test whether the method works correctly.
6. Write a method that returns the position of **the first occurrence** of an element from an array, such that it is greater than its two neighbors simultaneously. Otherwise the result must be **-1**.
7. Write a method that prints the digits of a given decimal number in a reversed order. For example **256**, must be printed as **652**.
8. Write a method that calculates the **sum of two very long positive integer numbers**. The numbers are represented as **array digits** and the last digit (the ones) is stored in the array at index 0. Make the method work for all numbers with length up to 10,000 digits.
9. Write a method that finds **the biggest element of an array**. Use that method to implement **sorting in descending order**.
10. Write a program that calculates and prints the **n!** for any **n** in the range **[1…100]**.
11. Write a program that solves the following tasks:
    - Put the digits from an integer number into a reversed order.
    - Calculate the average of given sequence of numbers.
    - Solve the linear equation **a * x + b = 0**.

    Create appropriate **methods** for each of the above tasks.

    Make the program show a **text menu** to the user. By choosing an option of that menu, the user will be able to choose which task to be invoked.

Perform validation of the input data:

- The integer number must be a positive in the range [1…50,000,000].
- The sequence of numbers cannot be empty.
- The coefficient **a** must be non-zero.

12. Write a method that calculates the sum of two polynomials with integer coefficients, for example **$(3x^2 + x - 3) + (x - 1) = (3x^2 + 2x - 4)$**.

13. * Write a method that calculates the product of two polynomials with integer coefficients, for example **$(3x^2 + x - 3) * (x - 1) = (3x^3 - 2x^2 - 4x + 3)$**.

Solutions and Guidelines

1. Use a method that takes the name as parameter of type **string**.

2. Use the expression **Max(a, b, c) = Max(Max(a, b), c)**.

   To **test the code** check whether the results from the invoked methods is correct for a set of examples that cover the most interesting cases, e.g. **Max(1,2)=2**; **Max(3,-1)=3**; **Max(-1,-1)=-1**; **Max(1,2,444444)=444444**; **Max(5,2,1)=5**; **Max(-1,6,5)=6**; **Max(0,0,0)=0**; **Max(-10,-10,-10)=-10**; **Max(2000000000,-2000000001,2000000002)=2000000002**; etc.

   You may write a **generic method** that works not just for **int** but for any other type **T** using the following declaration:

```
static T Max<T>(T a, T b) where T : IComparable<T> { … }
```

   Read more about the concept of **generic methods** in the section "Generic Methods" of chapter "Defining Classes".

   Instead of creating a program that checks whether the method works correctly, you can search in Internet for information about "**unit testing**" and **write unit tests** for your methods. You may also read about unit testing in the section "Unit Testing" of chapter "High-Quality Code".

3. Use the **reminder of division by 10** and then a **switch** statement.

4. The method must take as parameter an array of integer numbers (**int[]**) and the number that has to be counted (**int**). Test it with few examples like this: **CountOccurences(new int[]{3,2,2,5,1,-8,7,2}, 2) → 3**.

5. Just **perform a check**. The elements of the first and the last position in the array will be compared only with their left and right neighbor. Test examples like **GreaterThanNeighbours(new int[]{1,3,2}, 1) → true** and **GreaterThanNeighbours(new int[]{1}, 0) → true**.

6. Invoke the method from the **previous problem** in a **for**-loop.

7. There are two solutions:

First solution: Let the number is **num**. So while **num ≠ 0** we print its last digit (**num % 10**) and then divide **num** by 10.

Second solution: Convert the number into a string **string** and print it in a reverse order with a **for**-loop. This is a bit cheater's approach.

8. The reader must implement own method that **calculates the sum of very big numbers**. The digits on position zero will keep the ones; the digit on the first position will keep the tenths and so on. When two very big numbers are about to be calculated, the ones of their sum will be equal to **(firstNumber[0] + secondNumber[0]) % 10**, the tenths on other side will be equal to **(firstNumber[1] + secondNumber[1]) % 10 + (firstNumber[0] + secondNumber[0])/10** and so on.

9. First write a method that finds **the biggest element in array** and then modify it to find the biggest element in **given range of the array**, e.g. in the elements at indexes [3…10]. Finally find the **biggest number in the range [1…n-1]** and **swap it with the first element**, then find the biggest element in the range [2…n-1] and swap it with the second element of the array and so on. Think when the algorithm should finish.

10. The reader must implement own method that calculates the **product of very big numbers**, because the value of **100!** does not fit in variable of type **ulong** or **decimal**. The numbers can be represented in an array of reversed digits (one digit in each element). For example, the number **512** can be represented as **{2, 1, 5}**. Then the multiplication can be implemented in the way done in the elementary school (multiply digit by digit and then calculate the sum).

    Another easier way to work with extremely large numbers such as 100! is by using the library **System.Numerics.dll** (you have to add a reference to it in your project). Look for Information in internet about how to use the class **System.Numerics.BigInteger**.

    Finally calculate in a loop **k!** for **k** = **1**, **2**, …, **n**.

11. Firstly, create the necessary **methods**. To **create the menu** display a list in which the actions are represented as numbers (1 – reverse, 2 – average, 3 – equation). Ask the user to choose from 1 to 3.

12. **Use arrays to represent the polynomial** and the arithmetic rules that you know from math. For example the polynomial **($3x^2$ + x - 5)** can be represented as an array of the numbers **{-5, 1, 3}**. Bear in mind that it is useful at the **zero** position to put the coefficient for **x^0** (in our case -5), at the **first** position – the coefficient for **x^1** (in our case 1) and so on.

13. Use the instructions from the **previous task** and the rules for polynomial multiplication that you know from math. How to **multiple polynomials** can be read here: http://www.purplemath.com/modules/polymult.htm.

Chapter 10. Recursion

In This Chapter

In this chapter we are going to get familiar with **recursion and its applications**. Recursion represents a powerful programming technique in which a **method makes a call to itself** from within its own method body. By means of recursion we can solve complicated **combinatorial problems**, in which we can easily exhaust different combinatorial configurations, e.g. **generating permutations** and **variations** and simulating **nested loops**. We are going to demonstrate many examples of correct and incorrect usage of recursion and convince you how useful it can be.

What Is Recursion?

We call an object **recursive** if it contains itself, or if it is defined by itself.

Recursion is a programming technique in which **a method makes a call to itself** to solve a particular problem. Such methods are called **recursive**.

Recursion is a programming technique whose correct usage leads to elegant solutions to certain problems. Sometimes its usage could considerably simplify the programming code and its readability.

Example of Recursion

Let's consider the **Fibonacci numbers**. These are the elements of the following sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Each element of the sequence is formed by the sum of the previous two elements. The first two elements are equal to 1 by definition, i.e. the next two rules apply:

$F_1 = F_2 = 1$

$F_i = F_{i-1} + F_{i-2}$ (for $i > 2$)

Proceeding directly from the definition, we can implement the following **recursive method for finding the n^{th} Fibonacci number**:

```
static long Fib(int n)
{
  if (n <= 2)
  {
```

```
        return 1;
    }
    return Fib(n - 1) + Fib(n - 2);
}
```

This example shows how simple and natural the implementation of a solution can be when using recursion.

On the other hand, it can serve as an example of how attentive we have to be while programming with recursion. Although it is intuitive, the present solution is one of the **classical examples when the usage of recursion is highly inefficient** as there are many excessive calculations (of one and the same element of the sequence) due to the recursive calls.

We are going to consider the advantages and the disadvantages of using recursion later in this chapter.

Direct and Indirect Recursion

When in the body of a method there is a call to the same method, we say that the method is **directly recursive**.

If method A calls method B, method B calls method C, and method C calls method A we call the methods A, B and C **indirectly recursive** or **mutually recursive**.

Chains of calls in indirect recursion can contain multiple methods, as well as branches, i.e. in the presence of one condition one method to be called, and provided a different condition another to be called.

Bottom of Recursion

When using recursion, we have to be totally sure that after a certain count of steps we get a concrete result. For this reason we should have one or more cases in which the solution could be found directly, without a recursive call. These cases are called **bottom of recursion**.

In the example with Fibonacci numbers the bottom of recursion is when n is less than or equal to 2. In this base case we can directly return result without making recursive calls, because by definition the first two elements of the sequence of Fibonacci are equal to 1.

If a recursive method has no base case, i.e. bottom, it will become **infinite** and the result will be **`StackOverflowException`**.

Creating Recursive Methods

When we create recursive methods, it is necessary that we break the task we are trying to solve in **subtasks**, for the solution of which we can use the

same algorithm (recursively). The combination of solutions of all subtasks should lead to the solution of the initial problem.

In each recursive call the problem area should be limited so that at some point the **bottom of the recursion** is reached, i.e. breaking of each subtask must lead eventually to the bottom of the recursion.

Recursive Calculation of Factorial

The usage of recursion we will illustrate with a classic example – recursive calculation of factorial.

Factorial of **n** (written **n!**) is the product of all integers between 1 and **n** inclusive. By definition 0! = 1.

n! = 1.2.3…n

Recurrent Definition

When creating our solution, it is much more convenient to use the corresponding recurrent definition of factorial:

n! = 1, for n = 0

n! = n.(n-1)!, for n>0

Finding a Recurrent Dependence

The presence of recurrent dependence is not always obvious. Sometimes we have to find it ourselves. In our case we can do this by analyzing the problem and calculating the values of the factorial for the first few integers.

```
0! = 1
1! = 1 = 1.1 = 1.0!
2! = 2.1 = 2.1!
3! = 3.2.1 = 3.2!
4! = 4.3.2.1 = 4.3!
5! = 5.4.3.2.1 = 5.4!
```

From here you can easily see the recurrent dependability:

```
n! = n.(n-1)!
```

Algorithm Implementation

The bottom of our recursion is the simplest case n = 0, in which the value of the factorial is 1.

In the other cases we solve the problem for n-1 and multiply the result by n. Thus after a certain count of steps we are definitely going to reach the bottom

of the recursion, because between 0 and n there is a certain count of integer numbers.

Once we have these substantial conditions we can write a method, which computes factorial:

```
static decimal Factorial(int n)
{
   // The bottom of the recursion
   if (n == 0)
   {
      return 1;
   }
   // Recursive call: the method calls itself
   else
   {
      return n * Factorial(n - 1);
   }
}
```

By using this method we can create an application, which reads an integer from the console computes its factorial and then prints the obtained value:

RecursiveFactorial.cs

```
using System;

class RecursiveFactorial
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      decimal factorial = Factorial(n);
      Console.WriteLine("{0}! = {1}", n, factorial);
   }

   static decimal Factorial(int n)
   {
      // The bottom of the recursion
      if (n == 0)
      {
         return 1;
      }
      // Recursive call: the method calls itself
```

```
        else
        {
            return n * Factorial(n - 1);
        }
    }
}
```

Here is what the result of the execution of the application would be like if we enter 5 for n:

```
n = 5
5! = 120
```

Recursion or Iteration?

The calculation of factorial is often given as an example when explaining the concept of recursion, but in this case, as in many others, recursion is not the best approach.

Very often, if we are given a recurrent definition of the problem, the **recurrent** solution is intuitive and not posing any difficulty, while **iterative** (consecutive) solution is not always obvious.

In this particular case the implementation of the iterative solution is as short and simple, but is a bit **more efficient**:

```
static decimal Factorial(int n)
{
    decimal result = 1;

    for (int i = 1; i <= n; i++)
    {
        result = result * i;
    }

    return result;
}
```

We are going to consider the advantages and disadvantages of using recursion and iteration later in this chapter.

For the moment we should remember that before proceeding with recursive implementation we should think about an iterative variant, after which we should choose the better solution according to the situation.

Let’s look at another example where we could use recursion to solve the problem. This time we are going to consider an iterative solution, too.

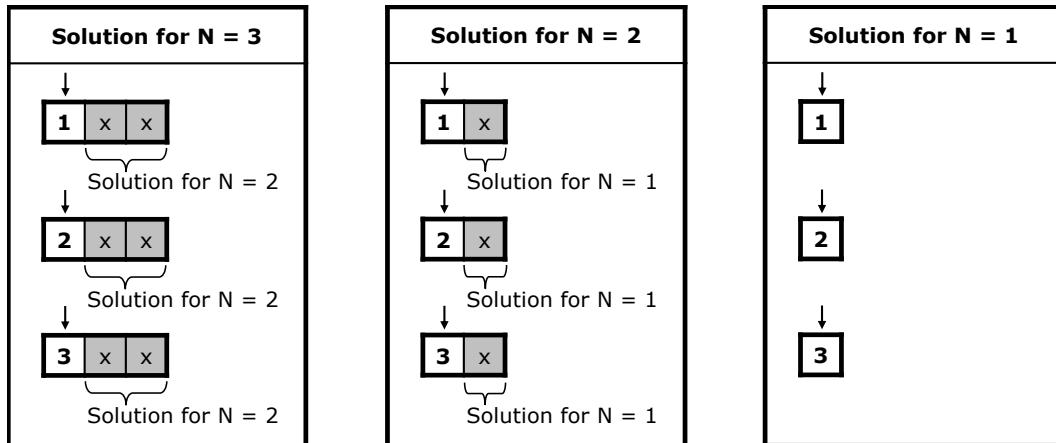
Simulation of N Nested Loops

Very often we have to write **nested loops**. It is very easy when they are two, three or any number previously assigned. However, if their count is not known in advance, we have to think of an alternative approach. This is the case with the following task.

Write a program that simulates the execution of **N nested loops** from **1** to **K**, where **N** and **K** are entered by the user. The result of the performance of the program should be equivalent to the execution of following fragment:

```
for (a1 = 1; a1 <= K; a1++)
  for (a2 = 1; a2 <= K; a2++)
    for (a3 = 1; a3 <= K; a3++)
      …
        for (aN = 1; aN <= K; aN++)
          Console.WriteLine("{0} {1} {2} … {N}",
            a1, a2, a3, …, aN);
```

For example, when N = 2 and K = 3 (which is equivalent to 2 nested loops from 1 to 3) and when N = 3 and K = 3, the results would be as follows:

```
               1 1                   1 1 1
               1 2                   1 1 2
               1 3                   1 1 3
  N = 2        2 1      N = 3        1 2 1
  K = 3   ->   2 2      K = 3   ->   …
               2 3                   3 2 3
               3 1                   3 3 1
               3 2                   3 3 2
               3 3                   3 3 3
```

The algorithm for solving this problem is not as obvious as in the previous example. Let’s consider two different solutions – one **recursive**, and one **iterative**.

Each row of the result can be regarded as ordered sequence of N numbers. The first one represents the current value of the counter of the loop, the second one – of the second loop, etc. On each position we can have value between 1 and K. The solution of our task boils down to finding all ordered sequences of N elements for N and K given.

Nested Loops – Recursive Version

If we are looking for a recursive solution to the problem, the first problem we are going to face is finding a recurrent dependence. Let’s look more carefully at the example from the assignment and put some further consideration.

Notice that, if we have calculated the answer for N = 2, then the answer for N = 3 can be obtained if we put on the first position each of the values of K (in this case from 1 to 3), and on the other two positions we put each of the couples of numbers, produced for N = 2. We can check that this rule applies for numbers greater than 3.

This way we have obtained the following dependence – starting from the first position, we put on the current position each of the values from **1** to **K** and continue **recursively** with the next position. This goes on until we reach position N, after which we print the obtained result (**bottom of the recursion**). Here is how the method looks implemented in C#:

```
static void NestedLoops(int currentLoop)
{
   if (currentLoop == numberOfLoops)
   {
      PrintLoops();
      return;
   }

   for (int counter=1; counter<=numberOfIterations; counter++)
   {
      loops[currentLoop] = counter;
      NestedLoops(currentLoop + 1);
   }
}
```

We are going to keep the sequence of values in an array called `loops`, which would be printed on the console by the method **`PrintLoops()`** when needed.

The method **`NestedLoops(…)`** takes one parameter, indicating the position in which we are going to place values.

In the loop we place consecutively on the current position each of the possible values (the variable **numberOfIterations** contains the value of **K** entered by the user), after which we call recursively the method **NestedLoops(…)** for the next position.

The bottom of the recursion is reached when the current position becomes **N** (the variable **numberOfIterations** contains the value of **N**, entered by the user). In this moment we have values on all positions and we print the sequence.

Here is a complete implementation of the **recursive nested loops** solution:

RecursiveNestedLoops.cs

```
using System;

class RecursiveNestedLoops
{
	static int numberOfLoops;
	static int numberOfIterations;
	static int[] loops;

	static void Main()
	{
		Console.Write("N = ");
		numberOfLoops = int.Parse(Console.ReadLine());

		Console.Write("K = ");
		numberOfIterations = int.Parse(Console.ReadLine());

		loops = new int[numberOfLoops];

		NestedLoops(0);
	}

	static void NestedLoops(int currentLoop)
	{
		if (currentLoop == numberOfLoops)
		{
			PrintLoops();
			return;
		}

		for (int counter=1; counter<=numberOfIterations; counter++)
		{
			loops[currentLoop] = counter;
```

```
            NestedLoops(currentLoop + 1);
        }
    }

    static void PrintLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            Console.Write("{0} ", loops[i]);
        }
        Console.WriteLine();
    }
}
```

If we run the application and enter for **N** and **K** respectively 2 and 4 as follows, we are going to obtain the following result:

```
N = 2
K = 4
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4
4 1
4 2
4 3
4 4
```

In the **Main()** method we enter values for N and K, create an array in which we are going to keep the sequence of values, after which we call the method **NestedLoops(…)**, starting from the first position.

Notice that as a parameter of the array we give 0 because we keep the sequence of values in an array, and as we already know, counting of array elements starts from 0.

The method **PrintLoops()** iterates all elements of the array and prints them on the console.

Nested Loops – Iterative Version

For the implementation of an **iterative solution of the nested loops** we can use the following algorithm, which finds the next sequence of numbers and prints it at each iteration:

1. In the beginning on each position place the number 1.
2. Print the current sequence of numbers.
3. Increment with 1 the number on position N. If the obtained value is greater than K replace it with 1 and increment with 1 the value on position N – 1. If its value has become greater then K, too, replace it with 1 and increment with 1 the value on position N – 2, etc.
4. If the value on the first position has become greater than K, the algorithm ends its work.
5. Go on with step 2.

Below we propose a straightforward implementation of the described **iterative nested loops algorithm**:

IterativeNestedLoops.cs

```
using System;

class IterativeNestedLoops
{
  static int numberOfLoops;
  static int numberOfIterations;
  static int[] loops;

  static void Main()
  {
    Console.Write("N = ");
    numberOfLoops = int.Parse(Console.ReadLine());

    Console.Write("K = ");
    numberOfIterations = int.Parse(Console.ReadLine());

    loops = new int[numberOfLoops];

    NestedLoops();
  }

  static void NestedLoops()
  {
    InitLoops();
```

```
        int currentPosition;

        while (true)
        {
            PrintLoops();

            currentPosition = numberOfLoops - 1;
            loops[currentPosition] = loops[currentPosition] + 1;

            while (loops[currentPosition] > numberOfIterations)
            {
                loops[currentPosition] = 1;
                currentPosition--;

                if (currentPosition < 0)
                {
                    return;
                }
                loops[currentPosition] = loops[currentPosition] + 1;
            }
        }
    }

    static void InitLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            loops[i] = 1;
        }
    }

    static void PrintLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            Console.Write("{0} ", loops[i]);
        }
        Console.WriteLine();
    }
}
```

The methods **Main()** and **PrintLoops()** are the same as in the implementation of the recursive solution.

The **NestedLoops()** method is different. It now implements the algorithm for iterative solution of the problem and for this reason does not get any parameters, unlike in the recursive version.

In the very beginning of this method we call the method **InitLoops()**, which iterates the elements of the array and places in each position 1.

The steps of the algorithm we perform in an infinite loop, from which we are going to escape in an appropriate moment by ending the execution of the methods via the operator **return**.

The way we implement step 3 of the algorithm is very interesting. The verification of the values greater than K, their substitution with 1 and the incrementing with 1 the value on the previous position (after which we make the same verification for it too) we implement by using one `while` loop, which we enter only if the value is greater than K.

For this purpose we first replace the value of the current position with 1. After that the position before it becomes current. Next we increment the value on the new position with 1 and go back to the beginning of the loop. These actions continue until the value on the current position is not less than or equal to K (the variable **numberOfIterations** contains the value of K), which is when we escape the loop.

When the value on the first position becomes greater than K (this is the moment when we have to end the execution), on its place we put 1 and try to increment the value on the previous position. In this moment the value of the variable **currentPosition** becomes negative (as the first position of the array is 0) and we end the execution of the method using the operator **return**. This is the end of our task.

We can now test it whit N = 3 and K = 2, for example:

```
N = 3
K = 2
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2
```

Which is Better: Recursion or Iteration?

If the algorithm solving of the problem is recursive, the implementation of recursive solution can be much more readable and elegant than iterative solution to the same problem.

Sometimes defining equivalent algorithm is considerably more difficult and it is not easy to be proven that the two algorithms are equivalent.

In certain cases by using recursion we can accomplish **much simpler, shorter and easy to understand solutions**.

On the other hand, recursive calls can consume much **more resources** (CPU time and memory). On each recursive call in the stack new memory is set aside for arguments, local variables and returned results. If there are too many recursive calls, a stack overflow could happen because of lack of memory.

In certain situations the recursive solutions can be much **more difficult to understand** and follow than the relevant iterative solutions.

Recursion is powerful programming technique, but we have to think carefully before using it. If used incorrectly, it can lead to inefficient and tough to understand and maintain solutions.

If by using recursion we reach a simpler, shorter and easier for understanding solution, not causing inefficiency and other side effects, then we can prefer recursive solution. Otherwise, it is better to think of iteration.

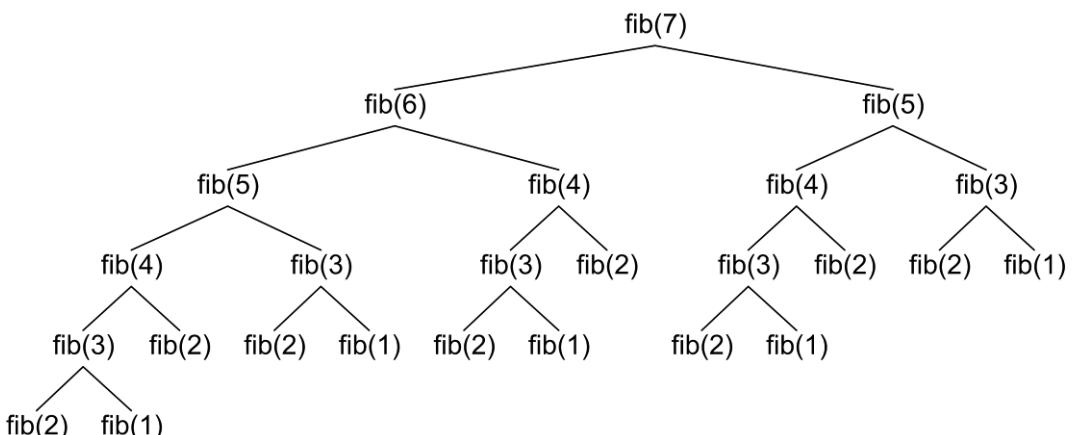
Fibonacci Numbers – Inefficient Recursion

Let's go back to the example with **finding the n^{th} Fibonacci number** and look more carefully at the recursive solution:

```
static long Fib(int n)
{
   if (n <= 2)
   {
      return 1;
   }
   return Fib(n - 1) + Fib(n - 2);
}
```

This solution is intuitive, short and easy to understand. At first sight it seems that this is a great example for applying recursion. The truth is that this is one of the classical examples of **inappropriate usage of recursion**. Let's run the following application:

RecursiveFibonacci.cs

```
using System;

class RecursiveFibonacci
```

```
{
    static void Main()
    {
        Console.Write("n = ");
        int n = int.Parse(Console.ReadLine());

        long result = Fib(n);
        Console.WriteLine("fib({0}) = {1}", n, result);
    }

    static long Fib(int n)
    {
        if (n <= 2)
        {
            return 1;
        }
        return Fib(n - 1) + Fib(n - 2);
    }
}
```

If we set the value of n = 100, the calculations would take so much time that no one would wait to see the result. The reason is that similar implementation is extremely inefficient. Each recursive call leads to two more calls and each of these calls causes two more calls and so on. That's why the tree of calls **grows exponentially** as shown on the figure below.

The count of steps for computing of **fib(100)** is of the order of 1.6 raised to the power 100 (this could be mathematically proven), whereas, if the solution is linear, the count of steps would be only 100.

The problem comes from the fact that there are a lot of excessive calculations. You can notice that **fib(2)** appears below many times on the **Fibonacci tree**:

Fibonacci Numbers – Efficient Recursion

We can **optimize the recursive method** for calculating the Fibonacci numbers by remembering (saving) the already calculated numbers in an array and making recursive call only if the number we are trying to calculate has not been calculated yet. Thanks to this small **optimization technique** (also known in computer science and dynamic optimization as **memoization** (not to be confused with **memorization**) the recursive solution would work for linear count of steps. Here is a sample implementation:

RecursiveFibonacciMemoization.cs

```
using System;

class RecursiveFibonacciMemoization
{
  static long[] numbers;

  static void Main()
  {
    Console.Write("n = ");
    int n = int.Parse(Console.ReadLine());

    numbers = new long[n + 2];
    numbers[1] = 1;
    numbers[2] = 1;

    long result = Fib(n);
    Console.WriteLine("fib({0}) = {1}", n, result);
  }

  static long Fib(int n)
  {
    if (0 == numbers[n])
    {
      numbers[n] = Fib(n - 1) + Fib(n - 2);
    }

    return numbers[n];
  }
}
```

Do you notice the difference? While with the initial version if n = 100 it seems like the computation goes on forever, with the optimized solution we get an answer instantly. As we will learn later in chapter "Algorithm Complexity", the first solution runs in **exponential time** while the second is **linear**.

```
n = 100
fib(100) = 3736710778780434371
```

Fibonacci Numbers – Iterative Solution

It is not hard to notice that we can solve the problem without using recursion, by calculating the Fibonacci numbers consecutively. For this purpose we are going to keep only the last two calculated elements of the sequence and use them to get the next element. Bellow you can see an implementation of the **iterative Fibonacci numbers calculation algorithm**:

IterativeFibonacci.cs

```
using System;

class IterativeFibonacci
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      long result = Fib(n);
      Console.WriteLine("fib({0}) = {1}", n, result);
   }

   static long Fib(int n)
   {
      long fn = 0;
      long fnMinus1 = 1;
      long fnMinus2 = 1;

      for (int i = 2; i < n; i++)
      {
         fn = fnMinus1 + fnMinus2;

         fnMinus2 = fnMinus1;
         fnMinus1 = fn;
      }

      return fn;
   }
}
```

This solution is as short and elegant, but does not hide risks of using recursion. Besides, it is efficient and does not require extra memory.

Concluding the previous examples we can give you the next recommendation:

	Avoid recursion, unless you are certain about how it works and what has to happen behind the scenes. Recursion is a great and powerful weapon, with which you can easily shoot yourself in the leg. Use it carefully!

If you follow this rule, you considerably will reduce the possibility of incorrect usage of recursion and the consequences, created by it.

More about Recursion and Iteration

Generally, when we have **a linear computational process**, we do not have to use recursion, because iteration can be constructed easily and leads to simple and **efficient calculations**. An example of linear computational process is the calculation of factorial. In it we calculate the elements of the sequence in which every next element depends only on the previous ones.

What is distinctive about the linear computational processes is that on each step of the calculating **recursion is called only once**, only in one direction. Schematically, a linear computational process we can describe as follows:

```
void Recursion(parameters)
{
  do some calculations;
  Recursion(some parameters);
  do some calculations;
}
```

In such a process, when we have only one recursive call in the body of the recursive method, it is not necessary to use recursion, because **the iteration is obvious**.

Sometimes, however, we have a **branched computational process** (like a tree). For example, the imitation of N nested loops cannot be easily replaced with iteration. You have probably noticed that our iterative algorithm, which imitates nested loops, works in a completely different principle. Try to implement the same without recursion and you will see it is not easy.

Ordinarily each recursion could **boil down to iteration by using a stack** of the calls (which is created through program execution), but this is complicated and there is no benefit from doing this. Recursion has to be used when it provides simple, easy-to-understand and efficient solution to a problem, for which we have no obvious iterative solution.

In **tree-like (branched) computational processes** on each step of the recursion a couple of recursive calls are made and the scheme of calculations

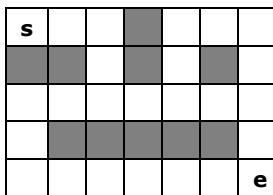
could be visualized as a **tree** (and not as a list like in linear calculations). For example, we saw what the tree of recursive calls would be like when we calculate the Fibonacci numbers.

A typical scheme of a tree computational process could be described with a pseudo-code in the following way:

```
void Recursion(parameters)
{
  do some calculations;
  Recursion(some parameters);
  …
  Recursion(some other parameters);
  do some calculations;
}
```

Tree computational processes could not be directly boiled down to recursive (unlike the linear processes). The case of Fibonacci is simple, because each next number is calculated via the previous, which we can calculate in advance. Sometimes, however, each next number is calculated not only via the previous, but via the next, and the recursive dependence is not so simple. In this case recursion turns out very efficient, if **implemented correctly** by avoiding duplicated calculations through **memoization**.

Use recursion for branched recursive calculations (and ensure each value is calculated only once). For linear recursive calculations prefer using iteration.

We are going to demonstrate the last statement with one classic example.

Searching for Paths in a Labyrinth – Example

We are given a **labyrinth** with a rectangular shape, consisting of N*M squares. Each square is either passable or impassable. An adventurer enters the labyrinth from its top left corner (there is the entrance) and has to reach the bottom right corner of the labyrinth (there is the exit). At each turn the adventurer can move up, down, left or right with one position and he has no right to go outside the binderies of the labyrinth, or step on impassable square. Passing through one and the same position is also forbidden (it is considered that the adventurer is lost if after a several turns he goes back to a position he has already been).

Write a computer program, which prints **all possible paths** from the beginning of the labyrinth to the exit.

This is a typical example of a problem, which can be easily solved with recursion, while with iteration the solution will be more complex and harder to implement.

Let's first draw an example in order to illustrate the problem and think about finding a solution:

You can see that there are **3 different paths** from the starting position to the end, which meets the requirements of the task (movement only on passable squares and not passing twice through any of the squares). Here you can see how these three paths look like:

On the figure above with numbers from 1 to 14 are marked the numbers of the corresponding turns of the paths.

Paths in a Labyrinth – Recursive Algorithm

How can we solve the problem? We can consider searching from a position in the labyrinth to the end of the labyrinth as a **recursive process** as follows:

- Let the current position in the labyrinth be (row, col). In the beginning we go from the **starting position** (0, 0).
- If the current position is the searched position (N-1, M-1), then we have **found a path** and we should print it.
- If the current position is **impassable**, we **go back** (we have no right to step on it).
- If the current position is already **visited**, we **go back** (we have no right to step on it twice.
- Otherwise, we **look for a path in four possible directions**. We search recursively (with the same algorithm) a path to the exit from the labyrinth by trying to go in all possible directions:
    - We try left: position (row, col-1).
    - We try up: position (row-1, col).
    - We try right: position (row, col+1).
    - We try down: position (row+1, col).

In order to reach this algorithmic solution we **think recursively**. We have the problem "searching for a path from given position to the exit". It can be boiled down to the following four sub problems:

- searching for a path from the position on the **left** from the current position to the exit;
- searching for a path from the position **above** the current position to the exit;
- searching for a path from the position on the **right** from the current position to the exit;
- searching for a path from the position **below** the current position to the exit.

If from each possible position, which we reach, we check the four possible directions and do not move in a circle (avoid passing through positions, on which we have already stepped on), we should find a path to the exit sooner or later (if such exists).

This time the recursion is not as simple as in the previous problems. On each step we have to check whether we have reached the exit and whether we are on a forbidden position; after that we should mark the position as visited and recursively call searching in the four directions. After returning from the recursive calls we have to mark as unvisited the starting point. In informatics such crawl is known as searching with **backtracking**.

Paths in a Labyrinth – Implementation

For the implementation of the algorithm we need to represent the labyrinth in a suitable way. We are going to use a two-dimensional array of characters, as in it we are going to mark with the character ' ' (space) the passable positions, with '**e**' the exit from the labyrinth and with '*****' the impassable positions. The starting position is marked as passable position. The positions we have already visited we are going to mark with the character '**s**'. Here is how the definition of the labyrinth is going to look like for our example:

```
static char[,] lab =
{
    {' ', ' ', ' ', '*', ' ', ' ', ' '},
    {'*', '*', ' ', '*', ' ', '*', ' '},
    {' ', ' ', ' ', ' ', ' ', ' ', ' '},
    {' ', '*', '*', '*', '*', '*', ' '},
    {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};
```

Let’s try to implement the recursive method for searching in a labyrinth. It should be something like this:

```
static char[,] lab =
{
    {' ', ' ', ' ', '*', ' ', ' ', ' '},
    {'*', '*', ' ', '*', ' ', '*', ' '},
    {' ', ' ', ' ', ' ', ' ', ' ', ' '},
    {' ', '*', '*', '*', '*', '*', ' '},
    {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

static void FindPath(int row, int col)
{
    if ((col < 0) || (row < 0) ||
        (col >= lab.GetLength(1)) || (row >= lab.GetLength(0)))
    {
        // We are out of the labyrinth
        return;
    }

    // Check if we have found the exit
    if (lab[row, col] == 'e')
    {
        Console.WriteLine("Found the exit!");
    }

    if (lab[row, col] != ' ')
    {
        // The current cell is not free
        return;
    }

    // Mark the current cell as visited
    lab[row, col] = 's';

    // Invoke recursion to explore all possible directions
    FindPath(row, col - 1); // left
    FindPath(row - 1, col); // up
    FindPath(row, col + 1); // right
    FindPath(row + 1, col); // down

    // Mark back the current cell as free
    lab[row, col] = ' ';
}

static void Main()
```

```
{
  FindPath(0, 0);
}
```

The implementation strictly follows the description from the above. In this case the size of the labyrinth is not stored in variables N and M, but is derived from the two-dimensional array lab, which stores the labyrinth: the count of the columns is **lab.GetLength(1)**, and the count of the rows is **lab.GetLength(0)**.

When entering the recursive method for searching, firstly we check if we go outside the labyrinth. In this case the searching is terminated, because going outside the boundaries of the labyrinth is forbidden.

After that we **check whether we have found the exit**. If we have, we print an appropriate message and the searching from the current position onward is terminated.

Next, we check if the current square is **available**. The square is available if the position is passable and we have not been on it on some of the previous steps (if it is not part of the current path from the starting position to the current cell of the labyrinth).

If the cell is available, we step on it. This is performed by marking it as visited (with the character '**s**'). After that we recursively search for a path in the four possible directions. After returning from the recursive search of the four possible directions, we step back from the current cell and mark it as available.

The **marking back** of the current position as available when leaving the current position is **substantial** because, when we go back, it is not a part of the current path. If we skip this action, not all paths to the exit would be found, but only some of them.

This is how the recursive method for searching for the exit from the labyrinth looks like. We should now only call the method from the **Main()** method, beginning the search from the starting position (0, 0).

If we run the program, we are going to see the following result:

```
Found the exit!
Found the exit!
Found the exit!
```

You can see that the exit has been found exactly three times. It seems that the algorithm works correctly. However, we are missing the printing of the path as a sequence of positions.

Paths in a Labyrinth – Saving the Paths

In order to print the paths we have found by our recursive algorithm, we can use an array, in which at every step we keep the direction taken (**L** – left, **U** – up, **R** – right, **D** – down). This array will keep in every moment the current path from the start of the labyrinth to the current position.

We are going to need an **array of characters** and a **counter** for the steps we have taken. The counter will keep how many times we have moved to the next position recursively, i.e. the current depth of recursion.

In order to work correctly, our program has to increment the counter when entering recursion and save the direction we have taken in the position in the array. When returning from a recursion, the counter should be reduced by 1. When an exit I found, the path can be printed (it consists of all the characters in the array from 0 to the position pointed by the counter).

What should be the **size of the array**? The answer to this question is easy; since we can enter one cell at most once, than the path would never be longer than the count of all cells (N*M). In our case the size of the maze is 7*5, i.e. the size of the array has to be 35.

Note: if you know the **List<T>** data structure is might be more appropriate to use **List<char>** instead of the array of chars. We will learn about lists in the chapter "Linear Data Structures".

This is an example implementation of the described idea:

```
static char[,] lab =
{
  {' ', ' ', ' ', '*', ' ', ' ', ' '},
  {'*', '*', ' ', '*', ' ', '*', ' '},
  {' ', ' ', ' ', ' ', ' ', ' ', ' '},
  {' ', '*', '*', '*', '*', '*', ' '},
  {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

static char[] path =
  new char[lab.GetLength(0) * lab.GetLength(1)];
static int position = 0;

static void FindPath(int row, int col, char direction)
{
  if ((col < 0) || (row < 0) ||
    (col >= lab.GetLength(1)) || (row >= lab.GetLength(0)))
  {
    // We are out of the labyrinth
    return;
  }
```

```
    // Append the direction to the path
    path[position] = direction;
    position++;

    // Check if we have found the exit
    if (lab[row, col] == 'e')
    {
        PrintPath(path, 1, position - 1);
    }

    if (lab[row, col] == ' ')
    {
        // The current cell is free. Mark it as visited
        lab[row, col] = 's';

        // Invoke recursion to explore all possible directions
        FindPath(row, col - 1, 'L'); // left
        FindPath(row - 1, col, 'U'); // up
        FindPath(row, col + 1, 'R'); // right
        FindPath(row + 1, col, 'D'); // down

        // Mark back the current cell as free
        lab[row, col] = ' ';
    }

    // Remove the last direction from the path
    position--;
}

static void PrintPath(char[] path, int startPos, int endPos)
{
    Console.Write("Found path to the exit: ");
    for (int pos = startPos; pos <= endPos; pos++)
    {
        Console.Write(path[pos]);
    }
    Console.WriteLine();
}

static void Main()
{
    FindPath(0, 0, 'S');
}
```

To make it easier we added one more parameter to the recursive method for searching path to the exit of the labyrinth: the direction we have taken to in order to reach the current position. This parameter has no meaning when going from the starting position. For this reason in the beginning we put a meaningless value '**S**'. After that, when printing, we skip the first element of the path.

If we start the program, we are going to get the three possible paths from the beginning to the end of the labyrinth:

```
Found path to the exit: RRDDLLDDRRRRRR
Found path to the exit: RRDDRRUURRDDDD
Found path to the exit: RRDDRRRRDD
```

Paths in a Labyrinth – Testing the Program

It seems like the algorithm works properly. It remains to test it with some more examples in order to make sure we have not made a stupid mistake. We can test the program with an empty labyrinth with size 1x1, with an empty labyrinth with size 3x3, or for instance with a labyrinth in which there is no path to the exit, and in the end with an enormous labyrinth, where there are a lot of paths.

If we run the tests, we are going to be convinced that in each case the program is working correctly.

Example input (labyrinth 1 x 1):

```
static char[,] lab =
{
  {'e'},
};
```

Example output:

```
Found path to the exit:
```

You can see that the output is correct, but the path is empty (with length 0), because the starting position coincides with the exit. We could improve the visualization in this case (for example print "**Empty path**"). Example input (empty labyrinth 3x3):

```
static char[,] lab =
{
  {' ', ' ', ' '},
  {' ', ' ', ' '},
  {' ', ' ', 'e'},
};
```

Example output for the above labyrinth:

```
Found path to the exit: RRDLLDRR
Found path to the exit: RRDLDR
Found path to the exit: RRDD
Found path to the exit: RDLDRR
Found path to the exit: RDRD
Found path to the exit: RDDR
Found path to the exit: DRURDD
Found path to the exit: DRRD
Found path to the exit: DRDR
Found path to the exit: DDRUURDD
Found path to the exit: DDRURD
Found path to the exit: DDRR
```

You can check that the output is correct – these are **all the paths** to the exit.

Let's try another example input (labyrinth 5x3 without a path to the exit):

```
static char[,] lab =
{
   {' ', '*', '*', ' ', ' '},
   {' ', ' ', ' ', '*', ' '},
   {'*', ' ', ' ', '*', 'e'},
};
```

Example output:

```
(there is no output)
```

You can see that the output is correct, but again we could add a more friendly message (for example "**No exit!**"), instead of any output.

Now we have to check what would happen when we have an **enormously big labyrinth**. Here is a sample input (labyrinth with size 15x9):

```
static char[,] lab =
{{' ','*',' ',' ',' ',' ','*',' ',' ',' ',' ','*','*',' ',' '},
 {' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ','*','*','*',' ','*',' ',' ',' ',' ',' ','*','*','*','*'},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ','e'}};
```

We run the program and it starts typing paths to the exit, but it **does not end** because there are **too many paths**. Here is how a small part of the output looks like:

```
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRURDRR
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURRD
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURDR
…
```

Now, let's try one last example – labyrinth with big size (15x9), in which there is no path to the exit:

```
static char[,] lab =
{
 {' ','*',' ',' ',' ',' ','*',' ',' ',' ',' ','*','*',' ',' '},
 {' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ','*','*','*',' ','*',' ',' ',' ',' ',' ','*','*','*','*'},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ','*','*',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ','*','e'},
};
```

We run the program and it **hangs, without printing anything**. It actually works very long for us to wait for it. It seems like there is a problem.

What is the problem? The problem is that the possible paths, analyzed by the algorithm are **too many** and their research takes **too much time**. Let's think how many these paths are. If a path to the exit is average 20 steps long and on each step there are 4 possible directions to be take, then 4^{20} paths have to be researched, which is a very big number. This evaluation of the count of possibilities is very inaccurate, but it gives orientation on the approximate order of possibilities.

What is the conclusion? The **backtracking** method **does not work**, when the variants are too many, and the fact they are too many can be easily concluded.

We are not going to torture you by making you find solution to the task. The problem of **searching all paths in a labyrinth has no efficient solution** for big labyrinths.

The problem has an efficient solution if it is formulated in a slightly different way: **find at least one exit from the labyrinth**. This task is far easier and

can be solved with one very small correction in the sample code: when escaping the recursion, we do not mark the current cell as available. This means to delete the following lines from the code:

```
// Mark back the current cell as free
lab[row, col] = ' ';
```

We can convince ourselves that after this change the program finds out very quickly if there is no path to the exit, and if there is, it very quickly finds one of them. It is not the shortest or longest, just the first path found.

Using Recursion – Conclusions

The general conclusion from the problem searching a path in a labyrinth is already formulated: **if you do not understand how recursion works, avoid using it**!

Be careful when you write recursive methods. Recursion is a **powerful programming technique** for solving combinatorial problems (problems in which we have to go through all variants), but **it is not for everyone**. We can easily make mistakes when using recursion. You may make the program "hang", or cause stack overflow with bottomless recursion. Always look for iterative solutions, unless you deeply understand how to use recursion.

As to the problem searching shortest path in a labyrinth you can solve it elegantly without recursion with the so called **BFS (breadth-first search)**, also known as **the wavefront algorithm**, which is elementary implemented with a queue. You can read more about the "**BFS**" algorithm in this article in Wikipedia: http://en.wikipedia.org/wiki/Breadth-first_search.

Exercises

1. Write a program to simulate **n nested loops** from **1** to **n**.

2. Write a program to generate **all variations with duplicates** of **n** elements class **k**. Sample input:

   ```
   n = 3
   k = 2
   ```

   Sample output:

   ```
   (1 1), (1 2), (1 3), (2 1), (2 2), (2 3), (3 1), (3 2), (3 3)
   ```

   Think about and implement an iterative algorithm for the same task.

3. Write a program to generate and print **all combinations with duplicates** of **k** elements from a set with **n** elements. Sample input:

```
n = 3
k = 2
```

Sample output:

```
(1 1), (1 2), (1 3), (2 2), (2 3), (3 3)
```

Think about and implement an iterative algorithm for the same task.

4. You are given a **set of strings**. Write a **recursive program**, which **generates all subsets**, consisting exactly **k** strings chosen among the elements of this set. Sample input:

```
strings = {'test', 'rock', 'fun'}
k = 2
```

Sample output:

```
(test rock), (test fun), (rock fun)
```

Think about and implement an **iterative algorithm** as well.

5. Write a **recursive program**, which prints **all subsets of a given set** of **N** words. Example input:

```
words = {'test', 'rock', 'fun'}
```

Example output:

```
(), (test), (rock), (fun), (test rock), (test fun),
(rock fun), (test rock fun)
```

Think about and implement an **iterative algorithm** for the same task.

6. Implement the **merge-sort algorithm recursively**. In it the initial array is divided into two equal in size parts, which are sorted (recursively via merge-sort) and after that the two sorted parts are merged in order to get the whole sorted array.

7. Write a recursive program, which generates and prints **all permutations of the numbers 1, 2, …, n**, for a given integer **n**. Example input:

```
n = 3
```

Example output:

```
(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

Try to find an **iterative solution** for generating permutations.

8. You are given an array of integers and a number **N**. Write a recursive program that finds **all subsets** of numbers in the array, which have a **sum N**. For example, if we have the array **{2, 3, 1, -1}** and **N=4**, we can obtain **N=4** as a sum in the following two ways: **4=2+3-1; 4=3+1**.

9. You are given an array of **positive** integers. Write a program that checks whether there is one or more numbers in the array **(subset), whose sum is equal to S**. Can you solve the task **efficiently for large arrays**?

10. You are given a **matrix** with passable and impassable cells. Write a recursive program that finds **all paths between two cells** in the matrix.

11. Implement the algorithm **BFS** (breadth-first search) for finding the **shortest path in a labyrinth**.

12. Modify the previous program to check **whether a path exists between two cells** without finding all possible paths. Test the program with a matrix 100x100 filled only with passable cells.

13. You are given a matrix with passable and impassable cells. Write a program that finds the **largest area of neighboring passable cells**.

14. Write a recursive program that **traverses the whole hard disk C:\ recursively** and prints all folders and files.

Solutions and Guidelines

1. Create a **recursive method Loops(int k)**, perform a **for**-loop from **1** to **n** and make a recursive call **Loops(k-1)** in the loop. The bottom of the recursion is when **k < 0**. Initially invoke **Loops(n-1)**.

2. The **recursive solution** is to modify the algorithm for **generating N nested loops**. In fact you need **k nested loops from 1 to n**.

   The **iterative solution** is as follows: start from the **first variation** in the lexicographical order: **{1, 1, …, 1} k times**. To **obtain the next variation**, **increase the last number**. If it becomes greater than **n**, change it to 1 and increase the next number on the left. Do the same on the left until the first number goes greater than **n**.

3. Modify the algorithms from **the previous problem** and always keep each number equal or greater than the number on the left of it. The easiest way to achieve this is to **generate k nested loops from 1 to n** and print only these combinations in which each number is greater or equal than the number on its left. You may optimize this approach to get generate directly an increasing sequence for better performance.

4. Let the strings' count be **n**. Use the implementation of **k nested loops** (recursive or iterative) with additional limitation that **each number is greater than the previous one**. Thus you will generate all different subsets of **k** elements in the range **[0…n-1]**. For each set consider the numbers from it as indices in the array of strings and print for each

number the corresponding string. For the example above, the set **{0, 2}** corresponds to the strings at position 0 and position 2, i.e. **(test, fun)**.

The **iterative algorithm** is similar to the iterative algorithm for generating **n nested loops**, but is more complicated because it needs to guarantee that each number is greater than the number on its left.

5. You can **use the previous task** and **call it N times** in order to generate consequently the empty set (**k**=0), followed by the all subsets with one element (**k**=1), all subsets with 2 elements (**k**=2), all subsets with 3 elements (**k**=3), etc.

   The problem has another **very smart iterative solution**: run a **loop from 0 to 2^N-1** and convert each of these numbers to **binary numeral system**. For example, for N=3 you will have the following binary representations of the numbers between 0 to 2^N-1:

```
000, 001, 010, 011, 100, 101, 110, 111
```

   Now for each binary representation take those words from the subset for which **have bit 1 on the corresponding position in the binary representation**. For instance, for the binary representation "101" take the first and the last string (at these positions there is 1) and omit the second string (at this position there is 0). Smart, isn't it?

6. In case you have any difficulties **search in Internet for "merge sort"**. You are going to find hundreds of implementations, including in C#. The challenge is to avoid allocating a new array for the result at each recursive call, because this is inefficient, and to **use only three arrays in the whole program**: two arrays to be merged merge and a third for the result from the merging. You will have to implement merging of two ranges of an array into a range of another array.

7. **Recursive algorithm**: suppose that the method **Perm(k)** permutes in all possible ways the elements of the array **p[]** at positions from **0** to **k-1** (inclusive). Firstly, initialize the array **p** with the numbers from 1 to N. **Implement recursively Perm(k)** in the following way:

   1. If **k == 0**, print the current permutation and exit the recursion (bottom of the recursion).
   2. Call **Perm(k-1)**.
   3. For each position **i** from **0 to k-1** do the following:
      a. Swap **p[i]** with **p[k]**.
      b. Recursively call **Perm(k-1)**.
      c. Swap back **p[i]** with **p[k]**.

   In the beginning call **Perm(n-1)** to start the recursive generation.

Iterative algorithm: read in Wikipedia how to generate from given permutation the next permutation in the lexicographic order iteratively: en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order.

8. The problem is not very different from the task with **finding all subsets among a given list of strings**. Shall it work fast enough with 500 numbers? Pay attention that we have to print **all subsets with sum N** which can be really big amount if N is very big and proper numbers exist in the array. For this reason **the task has no efficient solution**.

9. If we approach the problem by the method of generating of all possibilities, the solution **will not work for more than 20-30 numbers**. That's why we may approach it in a very different way in case the elements of the array are only positive, or are **limited in a certain range** (for example **[-50…50]**). Then we could use the following optimized algorithm based on **dynamic programming**:

   Assume we are given an array of numbers **p[]**. Let's denote by **possible(k, sum)** whether we could obtain **sum** by using only the numbers first **k** numbers (**p[0]**, **p[1]**, …, **p[k]**). Then, the following **recurrent dependencies** are valid:

   - **possible(0, sum) = true** if **p[0] == sum**
   - **possible(k, sum) = true** if **possible[k-1, sum] == true** or **possible[k-1, sum-p[k]] == true**

   The formula above shows that we can obtain **sum** from the elements of the array at positions **0** to **k** if one of the following two statements remains:

   - The element **p[k]** does not participate in the **sum** and the **sum** is obtained from the rest of the elements (from **0** to **k-1**);
   - The element **p[k]** participates in **sum** and the remainder **sum-p[k]** is obtained from the rest of the elements (from **0** to **k-1**).

   The implementation is not complex. Just calculate the recursive formulas by **recursive method**. We should be careful and not let already calculated values from the two-dimensional array **possible[,]** to be calculated twice. For this purpose we should keep for each possible **k** and **sum** the value **possible[k, sum]**. Otherwise the algorithm will not work for more than 20-30 elements.

   The regeneration of the numbers, which compose the found sum, may be implemented if we **go backwards from the sum n**, obtained from the first **k** numbers. At each step we examine how this sum can be obtained from the first **k–1** numbers (by taking the **k**th number or omitting it).

   Bear in mind that in the general case all possible sums of the numbers from the input array may be an awful lot. For instance, possible sums of 50 **int** numbers in the range [**Int32.MinValue** … **Int32.MaxValue**] are

enough so that we could not sum them in whatever data structure. If, however, all numbers in the input array are positive (as in our case), we could keep the sums in the range **[1…S]** because from the rest we could not obtain sum **S** by adding one or more numbers from the input array.

If the numbers in the input array are not mandatory positive, but are **limited in a range**, then all possible sums are limited in some range too and we could use the algorithm described above. For example, if the range of numbers is from -50 to 50, then the least sum is -50*S and the greatest is 50*S.

If the numbers in the input array are random and not limited in a range, then **the problem has no efficient solution**.

You could read more about this classical optimization problem in computer science called "**Subset Sum Problem**" in the following article in Wikipedia: http://en.wikipedia.org/wiki/Subset_sum_problem.

10. Follow the algorithms described in the section "Searching for Paths in a Labyrinth". Note that you need to find **all possible paths** (not just one of them) so don't expect your program to run fast for large input data.

11. Read the article about **BFS** in Wikipedia: http://en.wikipedia.org/wiki/Breadth-first_search. There are enough explanations and sample code. In order to implement a queue in C#, just an array or the .NET system class `System.Collections.Generics.Queue<T>`. For the elements of the queue you could use your own structure **Point**, containing **x** and **y** coordinates, or use two queues (one for each of the coordinates). You may also check the section BFS in the chapter "Trees and Graphs".

12. Follow the algorithms described in the section "Searching for Paths in a Labyrinth". You should run some graph traversal algorithm like **Depth-First Search (DFS)** or **Breath-First Search (BFS)**. You may read about them in Internet or check the sections about DFS and BFS in the chapter "Trees and Graphs". Your program should visit each cell at most once and should be fast, even on large matrices (like 1,000 x 1,000).

13. The same like the **previous exercise**: use DFS or BFS. By a recursive traversal or BFS traversal, find the areas of neighbor cells in the matrix one after another and mark each area's cells as visited. Do not visit again a visited cell. From all the areas found, remember the largest.

14. For each folder (starting from `C:\`) print the name and the files from the current folder and **call a recursion** for each subfolder. The problem is solved as example in the sections DFS and BFS in the chapter "Trees and Graphs". Your program may crash with `UnauthorizedAccessException` in case you do not have access permissions for some folders on the hard disk. This is typical for some Windows installations so you could start the traversal from another directory or catch the exception (see the "Catching Exceptions" section in the "Exception Handling" chapter).

Chapter 11. Creating and Using Objects

In This Chapter

In this chapter we are going to get familiar with the basic concepts of object-oriented programming – **classes and objects** – and we are going to explain how to use classes from the standard libraries of .NET Framework. We are going to mention some commonly used system classes and see how to **create and use** their instances (objects). We are going to discuss how we **can access fields** of an object, how to **call constructors** and how to work with static fields in classes. Finally, we are going to get familiar with the term "**namespaces**" – how they help us, how to include them and use them.

Classes and Objects

Over the last few decades programming and informatics have experienced incredible growth and concepts, which have changed the way programs, are built. **Object-oriented programming (OOP)** introduces such radical idea. We are going to make a short introduction to the principles of OOP and the concepts used in it. Firstly, we are going to explain what classes and objects are. These two terms are basic for OOP and inseparable part from the life of any modern programmer.

What Is Object-Oriented Programming?

Object-oriented programming (OOP) is a programming paradigm, which uses **objects** and their interactions for building computer programs. Thus an easy to understand, simple model of the subject area is achieved, which gives an opportunity to the programmer to solve intuitively (by simple logic) many of the problems, which occur in the real world.

For now we are not going to get into details what the goals and the advantages of OOP are, as well as explaining in details the principles for building hierarchies of classes and objects. We are going to mention only that programming techniques of OOP often include **encapsulation**, **abstraction**, **polymorphism** and **inheritance**. These techniques are out of the goals of the current chapter and we are going to consider them later in the chapter "Principles of Object-Oriented Programming". Now we will focus on **objects** as a basic concept in OOP.

What Is an Object?

We are going to introduce the concept **object** in the context of OOP. Software objects model real world objects or abstract concepts (which are also regarded as objects).

Examples of **real-world objects** are people, cars, goods, purchases, etc. abstract objects are concepts in an object area, which we have to model and use in a computer program. Examples of abstract objects are the data structures stack, queue, list and tree. They are not going to be a subject in this chapter, but we are going to see them in details in the next chapters.

In objects from the real world (as well as in the abstract objects) we can distinguish the following two groups of their characteristics:

- **States** – these are the characteristics of the object which define it in a way and describe it in general or in a specific moment
- **Behavior** – these are the specific distinctive actions, which can be done by the object.

Let's take for example an object from the real world – "dog". The states of the dog can be "name", "fur color" and "breed", and its behavior – "barking", "sitting" and "walking".

Objects in OOP combine data and the means for their processing in one. They correspond to objects in real world and contain data and actions:

- **Data members** – embedded in objects variables, which describe their states.
- **Methods** – we have already considered them in details. They are a tool for building the objects.

What Is a Class?

The **class** defines abstract characteristics of objects. It provides a structure for objects or a pattern which we use to describe the nature of something (some object). **Classes are building blocks of OOP** and are inseparably related to the **objects**. Furthermore, each object is an **instance** of exactly one specific class.

We are going to give as an **example a class and an object**, which is its instance. We have a **class `Dog`** and an **object `Lassie`**, which is an instance of the class **`Dog`** (we say it is an object of type **`Dog`**). The class **`Dog`** describes the characteristics of all dogs whereas **`Lassie`** is a certain dog.

Classes provide **modularity** in object-oriented programs. Their characteristics have to be meaningful in a common context so that they could be understood by people who are familiar with the problem area and are not programmers. For instance, the class **`Dog`** cannot have (or at least should not) a characteristic "RAM" because in the context of this class such characteristic has no meaning.

Classes, Attributes and Behavior

The class defines the **characteristics of an object** (which we are going to call **attributes**) and its **behavior** (actions that can be performed by the object). The attributes of the class are defined as its own variables in its body (called **member variables**). The behavior of objects is modeled by the definition of **methods** in classes.

We are going to illustrate the foregoing explanations through an **example of a real-world definition of a class**. Let's return to the example with the dog. We would like to define a class **Dog** that models the real object "dog". The class is going to include characteristics which are common for all dogs (such as breed and fur color), as well as typical for the dog behavior (such are barking, sitting, walking). In this case we are going to have attributes **breed** and **furColor**, and the behavior is going to be implemented by the methods **Bark()**, **Sit()** and **Walk()**.

Objects – Instances of Classes

From what has been said till now we know that each object is an instance of just one class and is created according to a pattern of this class. Creating the object of a defined class is called **instantiation** (creation). The **instance** is the object itself, which is created runtime.

Each object is in **instance** of a specific class. This instance is characterized by **state** – set of values, associated with class attributes.

In the context of such behavior the object consists of two things: current **state** and **behavior** defined in the class of the object. The state is specific for the instance (the object), but the behavior is common for all objects which are instances of this class.

Classes in C#

So far we have considered several common characteristics of OOP. A great part of the **modern programming languages are object-oriented**. Each of them has particular features for working with classes and objects. In this book we are going to focus only one of these languages – C#. It is good to know that the knowledge of OOP in C# would be useful to the reader no matter which object-oriented language he uses in practice. That is because **OOP is a fundamental concept in programming**, used by virtually all modern programming languages.

What Are Classes in C#?

A **class** in C# is defined by the keyword **class**, followed by an identifier (name) of the class and a set of data members and methods in a separate code block.

Classes in C# can contain the following elements:

- **Fields** – member-variables from a certain type;
- **Properties** – these are a special type of elements, which extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields. We are going to focus on them in the chapter "Defining Classes";
- **Methods** – they implement the manipulation of the data.

An Example Class

We are going to give an example of a class in C#, which contains the listed elements. The class **Cat** models the real-world object "cat" and has the properties **name** and **color**. The given class defines several fields, properties and methods, which we are going to use later. You can now see the definition of the class (we are not going to consider in details the definition of the classes – we are going to focus on that in the chapter "Defining Classes"):

```
public class Cat
{
  // Field name
  private string name;
  // Field color
  private string color;

  public string Name
  {
    // Getter of the property "Name"
    get
    {
      return this.name;
    }
    // Setter of the property "Name"
    set
    {
      this.name = value;
    }
  }

  public string Color
  {
    // Getter of the property "Color"
    get
    {
      return this.color;
    }
    // Setter of the property "Color"
```

```
        set
        {
            this.color = value;
        }
    }

    // Default constructor
    public Cat()
    {
        this.name = "Unnamed";
        this.color = "gray";
    }

    // Constructor with parameters
    public Cat(string name, string color)
    {
        this.name = name;
        this.color = color;
    }

    // Method SayMiau
    public void SayMiau()
    {
        Console.WriteLine("Cat {0} said: Miauuuuuu!", name);
    }
}
```

The example class **Cat** defines the **properties Name** and **Color**, which keep their values in the hidden (private) **fields name** and **color**. Furthermore, two **constructors** are defined for creating instances of the class **Cat**, respectively with and without parameters, and a **method** of the class **SayMiau()**.

After the example class is defined we can now use it in the following way:

```
static void Main()
{
    Cat firstCat = new Cat();
    firstCat.Name = "Tony";
    firstCat.SayMiau();

    Cat secondCat = new Cat("Pepy", "red");
    secondCat.SayMiau();
    Console.WriteLine("Cat {0} is {1}.",
        secondCat.Name, secondCat.Color);
}
```

If we execute the example, we are going to get the following output:

```
Cat Tony said: Miauuuuuu!
Cat Pepy said: Miauuuuuu!
Cat Pepy is Red.
```

We saw a simple example for defining and using classes, and in the section "Creating and Using Objects" we are going to explain in details how to create objects, how to access their properties and how to call their methods and this is going to allow us to understand how this example works.

System Classes

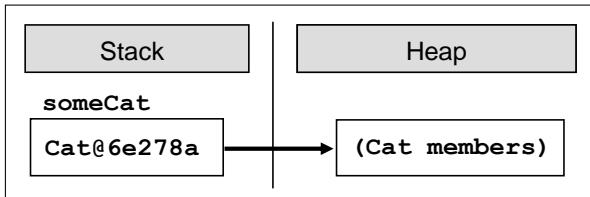
Calling the method **Console.WriteLine(…)** of the class **System.Console** is an example of usage of a **system class** in C#. We call system classes the classes defined in **standard libraries** for building applications with C# (or another programming language). They can be used in all our .NET applications (in particular those written in C#). Such are for example the classes **String**, **Environment** and **Math**, which we are going to consider later.

As we already know from chapter "Introduction to Programming" the **.NET Framework SDK** comes with a set of programming languages (like C# and VB.NET), compilers and **standard class library** which provides thousands of system classes for accomplishing the most common tasks in programming like console-based input / output, text processing, collection classes, parallel execution, networking, database access, data processing, as well as creating Web-based, GUI and mobile applications.

It is important to know that the implementation of the logic in classes is **encapsulated** (hidden) inside them. For the programmer it is important what they do, not how they do it and for this reason a great part of the classes is not publicly available (**public**). With system classes the implementation is often not available at all to the programmer. Thus, new **layers of abstraction** are created which is one of the basic principles in OOP.

We are going to pay special attention to system classes later. Now it is time to get familiar with creating and using objects in programs.

Creating and Using Objects

For now we are going to focus on **creating and using objects** in our programs. We are going to work with already defined classes and mostly with system classes from .NET Framework. The specificities of defining our own classes we are going to consider later in the chapter "Defining Classes".

Creating and Releasing Objects

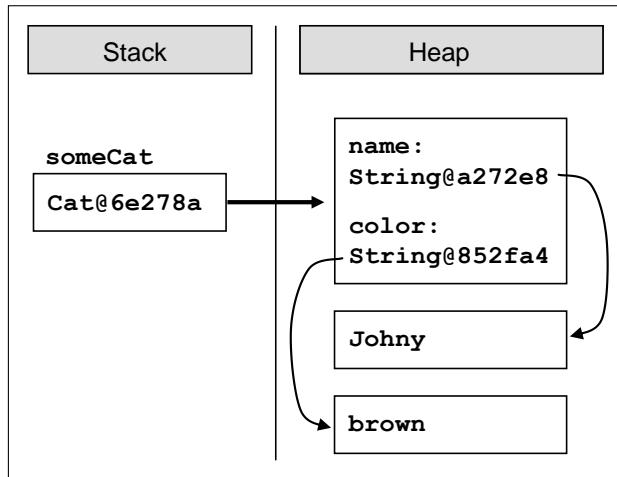
The creation of objects from preliminarily defined classes during program execution is performed by the **operator new**. The newly created object is usually assigned to the variable from type coinciding with the class of the

object (this, however, is not mandatory – read chapter "Principles of Object-Oriented Programming"). We are going to note that in this assignment the object is not copied, and only a **reference** to the newly created object is recorded in the variable (its address in the memory). Here is a simple example of how it works:

```
Cat someCat = new Cat();
```

The variable **someCat** of type **Cat** we assign the newly created **instance** of the class **Cat**. The variable **someCat** remains in the **stack**, and its value (the instance of the class **Cat)** remains in the **managed heap**:

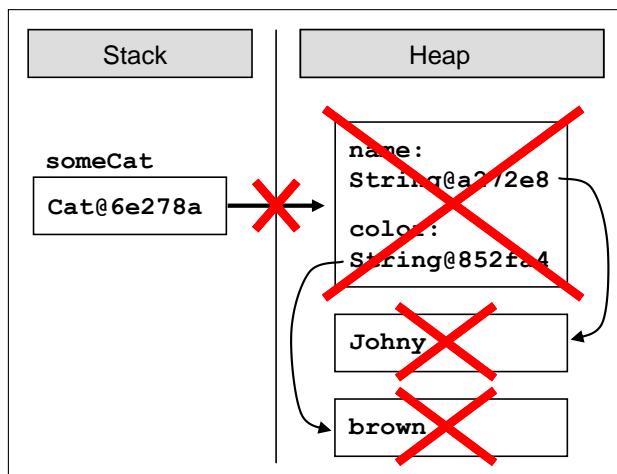
Creating Objects with Set Parameters

Now we are going to consider a slightly different variant of the example above in which we set parameters when creating the object:

```
Cat someCat = new Cat("Johnny", "brown");
```

In this case we would like the objects **someCat** to represent a cat whose name is "**Johnny**" and is brown. We indicate this by using the words "**Johnny**" and "**brown**", written in the brackets after the name of the class.

When creating an object with the operator **new**, two things happen: memory is set aside for this object and its data members are initialized. The **initialization** is performed by a special method called **constructor**. In the example above the initializing parameters are actually parameters of the constructor of the class.

We are going to discuss constructors after a while. As the member variables **name** and **color** of the class **Cat** are of reference type (of the class **String**), they are also recorded in the **dynamic memory (heap)** and in the object itself are kept their references (addresses / pointers).

The following figure illustrates how the **Cat** object is represented in the computer memory (arrows illustrated the **references** from one object to another):

Releasing the Objects

An important feature of working with objects in C# is that usually there is no need to manually destroy them and release the memory taken up by them. This is possible because of the embedded in .NET CLR system for cleaning the memory (**garbage collector**) which takes care of releasing unused objects instead of us. Objects to which there is no reference in the program at certain moment are **automatically released** and the memory they take up is released. This way many potential bugs and problems are prevented. If we would like to manually release a certain object, we have to destroy the reference to it, for example this way:

```
someCat = null;
```

This does not destroy the object immediately, but puts it in a state in which it is inaccessible to the program and the next time the garbage collector cleans the memory it is going to be released:

Access to Fields of an Object

The **access to the fields** and properties of a given object is done by the **operator .** (dot) placed between the names of the object and the name of the field (or the property). The operator **.** is not necessary in case we access field or property of given class in the body of a method of the same class.

We can **access** the **fields** and the **properties** either to extract data from them, or to assign new data. In the case of a property the access is implemented in exactly the same way as in the case of a field – C# give us this ability. This is achieved by the keywords **get** and **set** in the definition of the property, which perform respectively extraction of the value of the property and assignment of a new value. In the definition of the class **Cat** (given above) the properties are **Name** and **Color**.

Access to the Memory and Properties of an Object – Example

We are going to give an example of using a property of an object, as well as using the already defined above class **Cat**. We create an instance **myCat** of the class **Cat** and assign **"Alfred"** to the property **Name**. After that we print on the standard output a formatted string with the name of our cat. You can see an implementation of the example:

```
class CatManipulating
{
   static void Main()
   {
      Cat myCat = new Cat();
      myCat.Name = "Alfred";

      Console.WriteLine("The name of my cat is {0}.",
         myCat.Name);
   }
}
```

Calling Methods of Objects

Calling the methods of a given object is done through the **invocation operator ()** and with the help of the **operator .** (dot). The operator **dot** is not obligatory only in case the method is called in the body of another method of the same class. Calling a method is performed by its name followed by **()** or **(<parameters>)** for the case when we pass it some arguments. We already know how to invoke methods from the chapter "Methods".

Now is the moment to mention the fact that methods of classes have **access modifiers public**, **private** or **protected** with which the ability to call them could be restricted. We are going to consider these modifiers in the chapter "Defining Classes". For now it enough to know that the access modifier

public does not introduce any restrictions for calling the method, i.e. makes it publicly available.

Calling Methods of Objects – Example

We are going to complement the example we already gave as we call the method **SayMiau** of the class **Cat**. Here is the result:

```
class CatManipulating
{
    static void Main()
    {
        Cat myCat = new Cat();
        myCat.Name = "Alfred";

        Console.WriteLine("The name of my cat is {0}.",myCat.Name);
        myCat.SayMiau();
    }
}
```

After executing the program above the following text is going to be printed on the standard output:

```
The name of my cat is Alfred.
Cat Alfred said: Miauuuuuu!
```

Constructors

The **constructor** is a special method of the class, which is called automatically when **creating an object** of this class, and performs initialization of its data (this is its purpose). The constructor has no type of returned value and its name is not random, and mandatorily coincides with the class name. The constructor can be **with or without parameters**. A constructor without parameters is also called **parameterless constructor**.

Constructor with Parameters

The constructor can **take parameters** as well as any other method. Each class can have different count of constructors with one only restriction – the count and type of their parameters have to be different (different signature). When creating an object of this class, one of the constructors is called.

In the presence of several constructors in a class naturally occurs the question which of them is called when the object is created. This problem is solved in a very intuitive way as with methods. The appropriate constructor is chosen automatically by the compiler according to the given set of parameters when creating the object. We use the principle of the **best match**.

Calling Constructors – Example

Lets' take a look again at the definition of the class **Cat** and more particularly at the two constructors of the class:

```
public class Cat
{
   // Field name
   private string name;
   // Field color
   private string color;

   …

   // Parameterless constructor
   public Cat()
   {
      this.name = "Unnamed";
      this.color = "gray";
   }

   // Constructor with parameters
   public Cat(string name, string color)
   {
      this.name = name;
      this.color = color;
   }

   …
}
```

We are going to use these constructors to illustrate the usage of constructors with and without parameters. For the class **Cat** defined that way we are going to give an example of creating its instances by each of the two constructors. One of the objects is going to be an ordinary undefined cat, and the other – our brown cat Johnny. After that we are going to execute the method **SayMiau** for each of the cats and analyze the result. Source code follows:

```
class CatManipulating
{
   static void Main()
   {
      Cat someCat = new Cat();

      someCat.SayMiau();
      Console.WriteLine("The color of cat {0} is {1}.",
```

```
        someCat.Name, someCat.Color);

      Cat someCat = new Cat("Johnny", "brown");

      someCat.SayMiau();
      Console.WriteLine("The color of cat {0} is {1}.",
        someCat.Name, someCat.Color);
    }
}
```

As a result of the program's execution the following text is printed on the standard output:

```
Cat Unnamed said: Miauuuuuu!
The color of cat Unnamed is gray.
Cat Johnny said: Miauuuuuu!
The color of cat Johnny is brown.
```

Static Fields and Methods

The data members, which we considered up until, now implement **states of the objects** and are directly related to specific instances of the classes. In OOP there are special categories fields and methods, which are associated with the data type (class), and not with the specific instance (object). We call them **static members** because are independent of concrete objects. Furthermore, they are used without the need of creating an instance of the class in which they are defined. They can be fields, methods and constructors. Let's consider shortly static members in C#.

A static field or method in a given class is defined with the keyword **`static`**, placed before the type of the field or the type of returned value of the method. When defining a **static constructor**, the word static is placed before the name of the constructor. Static constructors are not going to be discussed in this chapter – for now we are going to consider only static fields and methods (the more curious readers can look up in MSDN).

When to Use Static Fields and Methods?

To find the answers of this question we have to understand very well the difference between static and non-static members. We are going to consider into details what it is.

We have already explained the main difference between the two types of members. Let's interpret the **class as a category of objects**, and **the object as a representative of this category**. Then the static members reflect the state and the behavior of the category itself, and the non-static the state and the behavior of the separate representatives of the category.

Now we are going to pay special attention to the **initialization of static and non-static fields**. We already know that non-static fields are initialized with the call to the constructor of the class when creating an instance of it – either inside the body of the constructor, or outside. However, the initialization of static fields cannot be performed when the object of the class is created, because they can be used without a created instance of the class. It is important to know the following:

	Static fields are initialized when the data type (the class) is used for the first time, during the execution of the program.

Now we shall see how to use static fields and methods in practice.

Static Fields and Methods – Example

The example, which we are going to give, solves the following simple problem: we need a method that every time returns a value greater with one than the value returned at the previous call of the method. We choose the first returned value to be 0. Obviously this method generates the sequence of natural number. Similar functionality is widely used in practice, for example, for uniform numbering of objects. Now we are going to see how this could be implemented with the means of OOP.

Let’s assume that the method is called **NextValue()** and is defined in a class called **Sequence**. The class has a field **currentValue** from type **int**, which contains the last returned value by the method. We would like the following two actions to be performed consecutively in the method body: the value of the field to be increased and its new value to be returned as a result. Obviously the returned by the method value does not depend on the concrete instance of the class **Sequence**. For this reason the method and the field are static. You can now see the described implementation of the class:

```
public class Sequence
{
  // Static field, holding the current sequence value
  private static int currentValue = 0;

  // Intentionally deny instantiation of this class
  private Sequence()
  {
  }

  // Static method for taking the next sequence value
  public static int NextValue()
  {
    currentValue++;
    return currentValue;
```

```
    }
}
```

The observant reader has noticed that the so defined class has a default constructor, which is declared as **private**. This usage of a constructor may seem strange, but is quite deliberate. It is good to know the following:

	A class that has only private constructors cannot be instantiated. Such class usually has only static members and is called "utility class".

For now we are not going to go into details about the **access modifiers public**, **private** and **protected**. We shall explain them comprehensively in the chapter "Defining Classes".

Let's take a look at a simple program, which uses the class **Sequence**:

```
class SequenceManipulating
{
    static void Main()
    {
        Console.WriteLine("Sequence[1...3]: {0}, {1}, {2}",
            Sequence.NextValue(), Sequence.NextValue(),
            Sequence.NextValue());
    }
}
```

The example prints on the standard output the first three natural numbers by triple consecutive call of the method **NextValue()** of the class **Sequence**. The result from this code is the following:

```
Sequence[1...3]: 1, 2, 3
```

If we try to create several different sequences, as the constructor of the class **Sequence** is declared **private**, we are going to get compile time error.

Examples of System C# Classes

After we got acquainted with the basic functionality of objects, we are going to consider briefly several **commonly used system classes** from the standard library of .NET Framework. This way we are going to see in practice the so far explained material, and also show how system classes ease our every-day work.

The System.Environment Class

We start with one of the basic system classes in .NET Framework: **System.Environment**. It contains a set of useful fields and methods, which

ease getting information about the hardware and the operating system, and some of them, give the ability to interact with the program environment. Here is a part of the functionality provided by this class:

- Information about the processors count, the computer network name, the version of the operating system, the name of the current user, the current directory, etc.
- Access to externally defined properties and environment variables, which we are not going to consider in this book.

Now we are going to show one interesting application of a method of the class **Environment**, which is commonly used in practice when developing programs with critical fast performance. We are going to detect the time needed for the execution of the source code with the help of the property **TickCount**. Here it is how it works:

```
class SystemTest
{
  static void Main()
  {
    int sum = 0;
    int startTime = Environment.TickCount;

    // The code fragment to be tested
    for (int i = 0; i < 10000000; i++)
    {
      sum++;
    }

    int endTime = Environment.TickCount;
    Console.WriteLine("The time elapsed is {0} sec.",
      (endTime - startTime) / 1000.0);
  }
}
```

The static property **TickCount** of the class **Environment** returns as a result the count of milliseconds that have passed since the computer is on until the time of the method call. With its help we detect the milliseconds past before and after the execution of the source code. Their difference is the wanted time for the execution of the fragment source code measured in milliseconds.

As a result of the execution of the program on the standard output we print the result of the following type (the measured time varies according to the current computer configuration and its load):

```
The time elapsed is 0.031 sec.
```

In the example we have used two static members of two system classes: the static property **Environment.TickCount** and the static method **Console.WriteLine(…)**.

The System.String Class

We have already met the **String** (**System.String**) class of .NET Framework, which represents strings. Let's recall that we can think of strings as a primitive data type in C#, although the work with them is different from the work with different primitive data types (integers, floating point numbers, Boolean variables, etc.). We are going to describe them in details in the chapter "Strings and Text Processing".

The System.Math Class

The **System.Math** class contains methods for performing basic **numeric and mathematical operations** such as raising a number to a power, taking a logarithm and square root, and some trigonometric functions. We are going to give a simple example, which illustrates its usage.

We want to make a program, which calculates the area of a triangle by given two sides and an angle between them in degrees. Therefore we need the method **Sin(…)** and the constant **PI** of the class **Math**. With the help of the **π** number we can easily convert to radians the entered in degrees angle. You can see an example implementation of the described logic:

```
class MathTest
{
  static void Main()
  {
    Console.WriteLine("Length of the first side:");
    double a = double.Parse(Console.ReadLine());
    Console.WriteLine("Length of the second side:");
    double b = double.Parse(Console.ReadLine());
    Console.WriteLine("Size of the angle in degrees:");
    int angle = int.Parse(Console.ReadLine());

    double angleInRadians = Math.PI * angle / 180.0;
    Console.WriteLine("Area of the triangle: {0}",
      0.5 * a * b * Math.Sin(angleInRadians));
  }
}
```

We can easily test the program if we check whether it calculates correctly the **area of an equilateral triangle**. For further convenience we choose the length of the side to be 2 – then we find the area with the well-known formula:

$$S = \frac{\sqrt{3}}{4} 2^2 = \sqrt{3} = 1{,}7320508 \ldots$$

We enter consecutively the numbers 2, 2, 60 and on the standard output we can see:

```
Face of the triangle: 1.73205080756888
```

Depending on your system localization (Region and Language Settings) your output might be "**1,73205080756888**" or "**1.73205080756888**". You might fix the decimal point to "**.**" by this line of code, executed at your program start:

```
System.Threading.Thread.CurrentThread.CurrentCulture =
  System.Globalization.CultureInfo.InvariantCulture;
```

The System.Math Class – More Examples

As we already saw, apart from mathematical methods, the **Math** class also defines two well known in mathematics constants: the trigonometric constant **π** and the Euler's number ***e***. Here is an example with them:

```
Console.WriteLine(Math.PI);
Console.WriteLine(Math.E);
```

When executing the code above, we get the following output:

```
3.141592653589793
2.718281828459045
```

The System.Random Class

Sometimes in programming we have to use **random numbers**. For instance, we would like to generate 6 random numbers in the range 1 to 49 (not necessarily unequal). This could be done by using the **System.Random** class and its method **Next()**. Before we use the **Random** class we have to create instance of it, at which point it is initialized with a random value (derived from the current system time in the operating system). After that we can randomly generate a number in the range **[0…n)** by calling the method **Next(n)**. Notice that this method can return zero, but always returns a random number smaller than the set value **n**. Therefore, if we would like to get a number in the range **[1…49]**, we have to use the expression **Next(49) + 1**.

Below is an example source code of a program, which generates 6 random numbers in the range from 1 to 49 by using the **Random** class (note that it is not guaranteed that the numbers are unique like in the classical Bulgarian lottery TOTO 6/49):

```
class RandomNumbersBetween1And49
{
  static void Main()
  {
    Random rand = new Random();
    for (int number = 1; number <= 6; number++)
    {
      int randomNumber = rand.Next(49) + 1;
      Console.Write("{0} ", randomNumber);
    }
  }
}
```

Here is how a possible output of the program looks like:

```
16 49 7 29 1 28
```

The System.Random Class – Generating a Random Password

To show you how useful **the random numbers generator** in .NET Framework can be, we are going to set as a task to **generate a random password** which is between 8 and 15 characters long, contains at least two capital letters, at least two small letters, at least one digit and at least three special chars. For this purpose we are going to use the following algorithm:

1. We start with an empty password. We create a generator of random numbers.
2. We generate twice a random capital letter and place it at a random position in the password.
3. We generate twice a random small letter and place it at a random position in the password.
4. We generate twice a random digit and place it at a random position in the password.
5. We generate three times a random special character and place it at a random position in the password.
6. Until this moment the password should consist of 8 characters. In order to supplement it to 15 characters at most, we can insert random count of times (between 0 and 7) at a random position in the password a random character (a capital letter, a small letter or a special char).

An implementation of the described algorithm is given below:

```
class RandomPasswordGenerator
{
  private const string CapitalLetters =
```

```
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private const string SmallLetters =
        "abcdefghijklmnopqrstuvwxyz";
    private const string Digits = "0123456789";
    private const string SpecialChars =
        "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
    private const string AllChars =
        CapitalLetters + SmallLetters + Digits + SpecialChars;

    private static Random rnd = new Random();

    static void Main()
    {
        StringBuilder password = new StringBuilder();

        // Generate two random capital letters
        for (int i = 1; i <= 2; i++)
        {
            char capitalLetter = GenerateChar(CapitalLetters);
            InsertAtRandomPosition(password, capitalLetter);
        }

        // Generate two random small letters
        for (int i = 1; i <= 2; i++)
        {
            char smallLetter = GenerateChar(SmallLetters);
            InsertAtRandomPosition(password, smallLetter);
        }

        // Generate one random digit
        char digit = GenerateChar(Digits);
        InsertAtRandomPosition(password, digit);

        // Generate 3 special characters
        for (int i = 1; i <= 3; i++)
        {
            char specialChar = GenerateChar(SpecialChars);
            InsertAtRandomPosition(password, specialChar);
        }

        // Generate few random characters (between 0 and 7)
        int count = rnd.Next(8);
        for (int i = 1; i <= count; i++)
        {
```

```
            char specialChar = GenerateChar(AllChars);
            InsertAtRandomPosition(password, specialChar);
        }

        Console.WriteLine(password);
    }

    private static void InsertAtRandomPosition(
        StringBuilder password, char character)
    {
        int randomPosition = rnd.Next(password.Length + 1);
        password.Insert(randomPosition, character);
    }

    private static char GenerateChar(string availableChars)
    {
        int randomIndex = rnd.Next(availableChars.Length);
        char randomChar = availableChars[randomIndex];
        return randomChar;
    }
}
```

Let’s explain several unclear moments in the source code. Let’s start from the definition of the constants:

```
private const string CapitalLetters =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
private const string SmallLetters =
    "abcdefghijklmnopqrstuvwxyz";
private const string Digits = "0123456789";
private const string SpecialChars =
    "~!@#$%^&*()_+=`{}[]\\|':;.,/?<>";
private const string AllChars =
    CapitalLetters + SmallLetters + Digits + SpecialChars;
```

Constants in C# are immutable variables whose values are assigned during their initialization in the source code of the program and after that they cannot be changed. They are declared with the modifier **const**. They are used for defining a number or a string, which afterwards is used many times in the program. This way repetition of certain values in the code is avoided and these values can be easily altered by changing only one place in the code. For example, if in a certain moment we decide that the character "**,**" (comma) should not be used when generating a password, we can change only one row in the program (the corresponding constant) and the change is going to reflect on every row where the constant is being used. In C# constants are

written in Pascal Case (the words in the name, merged together, each of them starts with an uppercase letter, and the rest of them are lowercase). More about constants we will learn in the section "Constants" in the chapter "Defining Classes".

Let's explain how the other parts of the program work. In the beginning, as a static member variable in the class **RandomPasswordGenerator** is created the random number generator **rnd**. As this variable **rnd** is defined in the class (not in the **Main()** method), it is accessible by the whole class (by each of its methods), and as it is defined static, it is accessible by the static methods, too. Thus, anywhere the program needs a random integer variable the same random number generator is used. It is initialized when the class **RandomPasswordGenerator** is loaded.

The method **GenerateChar()** returns a randomly chosen character in a set of characters given as a parameter. It works very simply: it chooses a random position in the set of characters (between 0 and the count of characters minus 1) and returns the characters at this position.

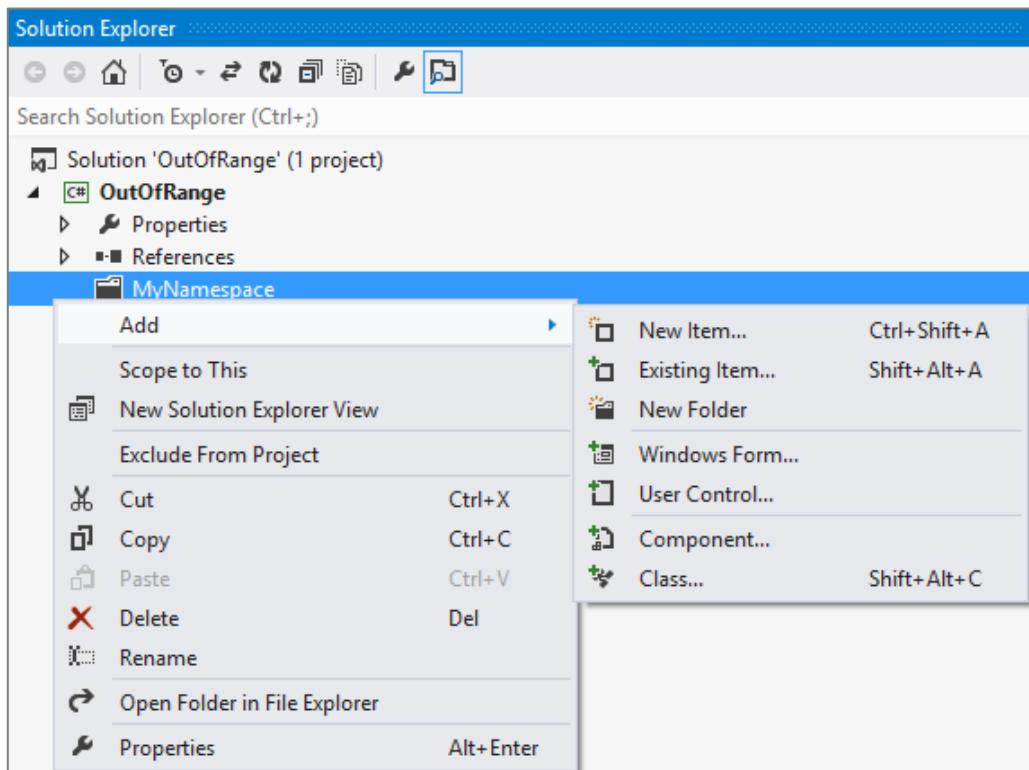
The method **InsertAtRandomPosition()** is not complicated too. It chooses a random position in the **StringBuilder** object, which is passed and inserts on this position the returned character. We are going to pay special attention to the class **StringBuilder** in the chapter "Strings and Text Processing".

Here is a sample output of the program for generating passwords, which we just considered (this output is different at each program run due to its randomness by nature):

```
8p#Rv*yTl{tN4
```

Namespaces

Namespace (package) in OOP we call a **container for a group of classes**, which are united by a common feature or are used in a common context. The namespaces contribute to a better logical organization of the source code by creating a semantic division of the classes in categories and makes easier their usage in the source code. Now we are going to consider namespaces in C# and are going to see how we can use them.

What Are Namespaces in C#?

Namespaces in C# are **named groups of classes**, which are logically related without any specific requirement on how to be placed in the file system. However, it is considered that the folder name should match the namespace name and the names of the files should match the names of the classes, which are defined in them. We have to note that in some programming languages the compilation of the source code in a given namespace depends on the distribution of the elements of the namespace in folders and files on the disk. In Java, for instance, the described file

organization is mandatory (if it is not followed, compilation errors occur). C# is not so strict regarding this.

Now, let's consider the mechanism for defining namespaces.

Defining Namespaces

In case we like to create a new namespace or a new class which belongs to a given namespace, in Visual Studio this happens automatically by the commands in the context menu of the Solution Explorer (on right click on the corresponding folder). By default the Solution Explorer is visualized like a Dock in the right part of the integrated environment. We are going to illustrate how we could add a new class in the already existing namespace **MyNamespace** by the context menu of Solution Explorer in Visual Studio:

As the project is called **MyConsoleApplication** and we are adding in its folder **MyNamespace**, the newly created class is going to be in the following namespace:

```
namespace MyConsoleApplication.MyNamespace
```

If we have defined a class in its own file and we like to add it in a new or already existing namespace, it is not hard to do it manually. It is enough to change the named block with a keyword **namespace** in the class:

```
namespace <namespace_name>
{
  …
}
```

In the definition we use the keyword **namespace**, followed by the full name of the namespace. It is considered that the namespaces in C# start with a capital letter and are written in Pascal Case. For example, if we have to make a namespace containing classes for string processing, it is desirable we name it **StringUtils**, and not **string_utils**.

Nested Namespaces

Except classes, **namespaces can contain other namespaces** in themselves (nested namespaces). This way, intuitively we create a hierarchy of namespaces, which allows even more precise distribution of classes according to their semantics.

When naming namespaces in the hierarchy we use the character **.** as a separator (dot notation). For example, the namespace **System** from .NET Framework contains in itself the sub-namespace **Collections** and thus the full name of the nested namespace **Collections** is **System.Collections**.

Full Names of Classes

In order to absolutely understand the meaning of namespaces, it is important for us to know the following:

Classes are required to have unique names only within the namespaces, in which they are defined.

Outside a given namespace we can have classes with random names regardless of whether they match with any of the names of classes in the namespace. This is because classes in the namespace are uniquely defined in its context. It is time to see how to define syntactically this uniqueness.

Full name of the class we call the first name of the class, preceded by the name of the namespace in which it is defined. The full name of each class is unique. Again we use dot notation:

```
<namespace_name>.<class_name>
```

Let's take, for example, the system class **CultureInfo**, defined in the namespace **System.Globalization** (we have already used it in the chapter "Console Input and Output"). According to the definition, the full name of the class is **System.Globalization.CultureInfo**.

In .NET Framework sometimes there are classes from different namespaces with matching names, for example:

```
System.Windows.Forms.Control
System.Web.UI.Control
System.Windows.Controls.Control
```

Inclusion of a Namespace

When building an application according to the object area, very often it is necessary to use the classes of a namespace multiple times. For the programmer's convenience there is a mechanism for **inclusion of a namespace** in the current file with a source code. After the given namespace is included, all classes defined in it may be used without the need to use their full names.

The inclusion of a namespace in the current source code file is executed with the **keyword `using`** in the following way:

```
using <namespace_name>;
```

We are going to pay attention to an important feature of including namespaces in the described way. All classes defined directly in the namespace **`<namespace_name>`** are included and can be used, but we have to know the following:

Inclusion of namespaces is not recursive, i.e. when including a namespace the classes from the nested namespaces are not included.

For example, the inclusion of namespaces **`System.Collections`** **does not** automatically include the classes from its nested namespace **`System.Collections.Generic`**. When used, either we have to apply their full names, or to include the namespace, which contains them.

Using a Namespace – Example

In order to illustrate the principle of inclusion of a namespace, we are going to consider the following program which reads numbers, saves them in lists and counts how many of them are integer numbers and how many are double:

```
class NamespaceImportTest
{
  static void Main()
  {
    System.Collections.Generic.List<int> ints =
      new System.Collections.Generic.List<int>();
```

```
    System.Collections.Generic.List<double> doubles =
      new System.Collections.Generic.List<double>();

    while (true)
    {
      int intResult;
      double doubleResult;
      Console.WriteLine("Enter an int or a double:");
      string input = Console.ReadLine();

      if (int.TryParse(input, out intResult))
      {
        ints.Add(intResult);
      }
      else if (double.TryParse(input, out doubleResult))
      {
        doubles.Add(doubleResult);
      }
      else
      {
        break;
      }
    }

    Console.Write("You entered {0} ints:", ints.Count);
    foreach (var i in ints)
    {
      Console.Write(" " + i);
    }
    Console.WriteLine();

    Console.Write("You entered {0} doubles:", doubles.Count);
    foreach (var d in doubles)
    {
      Console.Write(" " + d);
    }
    Console.WriteLine();
  }
}
```

For this purpose the program uses the class **System.Collections.Generic.List** as it calls it by its full name.

Let's see how the program above works: we enter consecutively the values **4**, **1.53**, **0.26**, **7**, **2**, **end**. We get the following result on the standard output:

```
You entered 3 ints: 4 7 2
You entered 2 doubles: 1.53 0.26
```

The program does the following: it gives the user the opportunity to enter consecutively numbers, which may be integer or double. This continues until the moment in which a value different from a number is entered. Then on the standard output two rows are displayed, respectively with integer and double numbers.

For the implementation of the described actions we use two helping objects respectively of type **System.Collections.Generic.List<int>** and **System.Collections.Generic.List<double>**. Obviously, the full names of the classes make the code unreadable, and cause inconveniences. We can easily avoid this effect by including the namespace **System.Collections.Generic** and use directly the classes by name. You can now see the shortened version of the program above:

```
using System.Collections.Generic;

class NamespaceImportTest
{
  static void Main()
  {
    List<int> ints = new List<int>();
    List<double> doubles = new List<double>();
    …
  }
}
```

Exercises

1. Write a program, which reads from the console a year and **checks if it is a leap year**.
2. Write a program, which generates and prints on the console **10 random numbers** in the range [100, 200].
3. Write a program, which prints, on the console **which day of the week is today**.
4. Write a program, which prints on the standard output the **count of days, hours, and minutes, which have passes since the computer is started** until the moment of the program execution. For the implementation use the class **Environment**.
5. Write a program which by given two sides **finds the hypotenuse of a right triangle**. Implement entering of the lengths of the sides from the

standard input, and for the calculation of the hypotenuse use methods of the class **Math**.

6. Write a program which **calculates the area of a triangle** with the following given:
   - three sides;
   - side and the altitude to it;
   - two sides and the angle between them in degrees.
7. Define your own namespace **CreatingAndUsingObjects** and place in it two classes **Cat** and **Sequence**, which we used in the examples of the current chapter. Define one more namespace and make a class, which calls the classes **Cat** and **Sequence**, in it.
8. Write a program which creates 10 objects of type **Cat**, gives them names **CatN**, where **N** is a unique serial number of the object, and in the end call the method **SayMiau()** for each of them. For the implementation use the namespace **CreatingAndUsingObjects**.
9. Write a program, which **calculates the count of workdays between the current date and another given date** after the current (inclusive). Consider that workdays are all days from Monday to Friday, which are not public holidays, except when Saturday is a working day. The program should keep a list of predefined public holidays, as well as a list of predefined working Saturdays.
10. You are given a **sequence of positive integer numbers** given as string of numbers separated by a space. Write a program, which **calculates their sum**. Example: "**43 68 9 23 318**" → **461**.
11. Write a program, which **generates a random advertising message** for some product. The message has to consist of laudatory phrase, followed by a laudatory story, followed by author (first and last name) and city, which are selected from predefined lists. For example, let's have the following lists:
    - **Laudatory phrases**: {"The product is excellent.", "This is a great product.", "I use this product constantly.", "This is the best product from this category."}.
    - **Laudatory stories**: {"Now I feel better.", "I managed to change.", "It made some miracle.", "I can't believe it, but now I am feeling great.", "You should try it, too. I am very satisfied."}.
    - **First name** of the author: {"Dayan", "Stella", "Hellen", "Kate"}.
    - **Last name** of the author: {"Johnson", "Peterson", "Charls"}.
    - **Cities**: {"London", "Paris", "Berlin", "New York", "Madrid"}.

    Then the program would print randomly generated advertising message like the following:

```
I use this product constantly. You should try it, too. I am
very satisfied. -- Hellen Peterson, Berlin
```

12. * Write a program, which calculates the value of a given numeral expression given as a string. The numeral expression consists of:
    - real numbers, for example **5**, **18.33**, **3.14159**, **12.6**;
    - arithmetic operations: **+**, **-**, *****, **/** (with their standard priorities);
    - mathematical functions: **ln(x)**, **sqrt(x)**, **pow(x, y)**;
    - brackets for changing the priorities of the operations: **(** and **)**.

Note that the numeral expressions have priorities, for example the expression **-1 + 2 + 3 * 4 - 0.5 = (-1) + 2 + (3 * 4) - 0.5 = 12.5**.

Solutions and Guidelines

1. Use **DateTime.IsLeapYear(year)**.
2. Use the class **Random**. You may generate random numbers in the range [100, 200] by calling **Random.Next(100, 201)**.
3. Use **DateTime.Today.DayOfWeek**.
4. Use the property **Environment.TickCount**, in order to get the count of passed milliseconds. Use the fact that one second has 1,000 milliseconds; one minute has 60 seconds; one hour has 60 minutes and one day has 24 hours.
5. The hypotenuse of a rectangular triangle could be found with the **Pythagorean Theorem** $\mathbf{a}^2 + \mathbf{b}^2 = \mathbf{c}^2$, where **a** and **b** are the two sides, and **c** is the hypotenuse. Take square root of the two sides of the equation in order to get the length of the hypotenuse. Use the **Sqrt(…)** methods of the **Math** class.
6. For the first sub-problem of the task use the **Heron's Formula** $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = \frac{a+b+c}{2}$. For the second sub-problem use the **formula**: $S = \frac{a*h_a}{2}$. For the third sub-problem use the **formula**: $S = \frac{a * b * sin(\gamma)}{2}$. For the sine use the **System.Math** class.
7. Make a **new project in Visual Studio**, right click on the folder and choose the menu **Add → New Folder**. Then enter the name of the folder and press [Enter], right click on the newly made folder and choose **Add → New Item…** from the list choose **Class**, for the name of the new class enter **Cat** and press [Add]. Change the definition of the newly created class with the definition, which we gave to this chapter, to put the classes in a **namespace**. Make the same to the class **Sequence**.

8. Create an array with 10 elements of type **Cat**. Create 10 objects of type **Cat** in a loop (use a constructor with parameters) and assign them to the corresponding element of the array. For the serial number of the objects use the method **NextValue()** of the **Sequence** class. In the end again in an array use the method **SayMiau()** for each of the array elements.

9. Use the class **System.DateTime** and the methods in it. You can execute a loop from the current date (**DateTime.Now.Date**) to the end date, consecutively incrementing the day by the method **AddDays(1)** and count the working days according to your country (e.g. all days except Saturday and Sunday and a few fixed non-working official holidays).

   Another approach that might work is to **subtract the dates** to find the **TimeSpan** between them (**DateTime** values can be subtracted, just like a numbers). This will give you the count of days between the dates. You will need to perform some additional calculations to find how much weekends are included in this count and discard them.

10. Use **String.Split(' ')** to split the string by spaces. Then use **Int32.Parse(…)** to extract the separate numbers from the obtained **string** array as **int** values and sum them.

11. Use the class **System.Random** and its method **Next(…)** to select a random laudatory phrase, laudatory story, first name, last name and city and combine them.

12. **Calculating a numeral expression** is **quite hard** and is unlikely a beginner programmer to solve it correctly without external help. As a start check out the article in Wikipedia about the "**Shunting-yard algorithm**" (en.wikipedia.org/wiki/Shunting-yard_algorithm) describing how to convert an expression from to **postfix notation** (reversed Polish notation), and the article about **calculating a postfix expression** (en.wikipedia.org/wiki/Reverse_Polish_notation). There are really much special cases, so be sure to test your solution carefully.

Chapter 12. Exception Handling

In This Chapter

In this chapter we will discuss **exceptions** in the object-oriented programming and in C# in particular. We will learn how to **handle exceptions** using the **`try-catch`** construct, how to pass them to the calling methods and how to **throw standard or our own exceptions** using the **`throw`** construct. We will give various examples for using exceptions. We will look at the types of exceptions and the **exceptions hierarchy** in the .NET Framework. At the end, we will look at the advantages of using exceptions, best practices and how to apply them in different situations.

What Is an Exception?

When we write a program, we describe step-by-step what the computer must do (at least in imperative programming; in the functional programming things look a bit different) and in most of the cases we rely that the program will execute normally. Indeed, most of the time, programs are following this normal pattern, but there are some exceptions. Let's say we want to read a file and display its contents on the screen. Let's assume the file is located on a remote server and during the process of reading it, the connection goes down. The file then will be only partially loaded. The program will not be able to execute normally and show file's contents on the screen. In this case, we have an **exception** from the normal (and correct) program execution and this exception must be reported to the user and/or the administrator.

Exceptions

Exception is a **notification that something interrupts the normal program execution**. Exceptions provide a programming paradigm for detecting and reacting to unexpected events. When an exception arises, the state of the program is saved, the normal flow is interrupted and the control is passed to an **exception handler** (if such exists in the current context).

Exceptions are raised or **thrown** by programming code that must send a signal to the executing program about **an error or an unusual situation**. For example, if we try to open a file, which doesn't exist, the code responsible for opening the file will detect this and will throw an exception with a proper error message.

Exceptions are one of the main paradigms of object-oriented programming (OOP), which is described in details in the chapter "Object-Oriented Programming Principles".

Catching and Handling Exceptions

Exception handling is a mechanism, which allows **exceptions to be thrown and caught**. This mechanism is provided internally by the CLR (Common Language Runtime). Parts of the exception handling infrastructure are the **language constructs in C#** for throwing and catching exceptions. CLR takes care to propagate each exception to the code that can handle it.

Exceptions in the Object-Oriented Programming

In Object-Oriented Programming (OOP), exceptions are a powerful mechanism for **centralized processing of errors** and exceptional situations. This mechanism replaces the procedure-oriented method of error handling in which each function returns a code indicating an error or a successful execution.

Usually in OOP, a code executing some operation will cause an exception if there is a problem and the **operation could not be successfully completed**. The method causing the operation could catch the exception (and handle the error) or pass the exception through to the calling method. This allows handling errors to be delegated to some upper level in the call stack and in general, allows flexible management of errors and unexpected situations.

Another fundamental concept is **exceptions hierarchy**. In OOP, exceptions are classes and they can be inherited to build hierarchies. When an exception is handled (caught), the handling mechanism could catch a whole class of exceptions and not just a particular error (as in the traditional procedural programming).

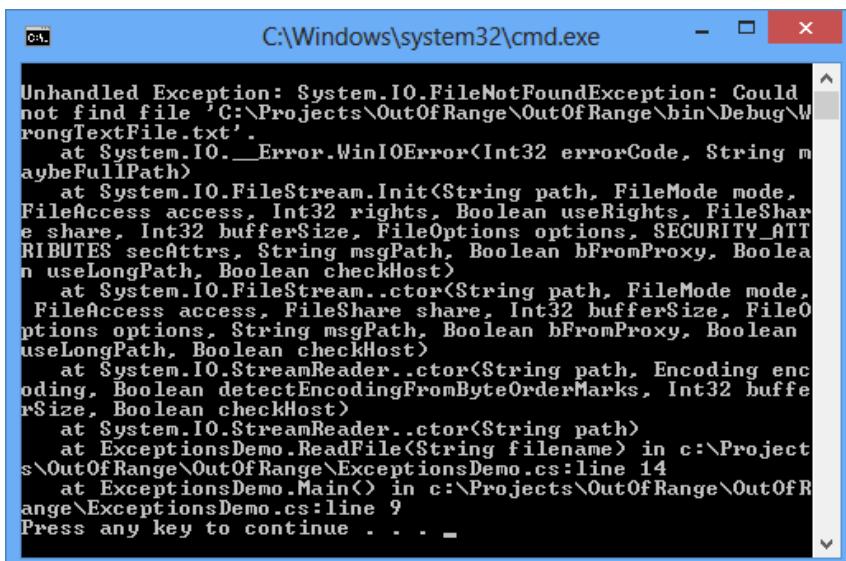
In OOP, it is recommended to **use exceptions for managing error situations or unexpected events** that may arise during a program execution. This replaces the procedural error-handling approach and gives important advantages such as centralized error processing, handling multiple errors in one place and ability to pass errors to a higher-level handler. Another important advantage is that exceptions self-describe themselves and can create hierarchies.

Sometimes exceptions are used not so much to signal a problem but to handle some expected event. This is not considered a good practice as exceptions should not control the normal flow of the program. At the end of the chapter we will look in more details into this.

Exceptions in .NET

Exception in .NET is an **object**, which signals an error or an event, which is not anticipated in the normal program flow. When such unusual event takes place, the executing method 'throws' a special object containing information

about the type of the error, the place in the program where the error occurred as well as the program state at the moment of the error.

Each exception in .NET contains the so-called **stack trace**, which gives information of where exactly the error occurred. This will be discussed in more details later in this chapter.

An Example Code Throwing an Exception

Here is an example for a code that will **throw an exception**:

```
class ExceptionsDemo
{
   static void Main()
   {
      string fileName = "WrongTextFile.txt";
      ReadFile(fileName);
   }

   static void ReadFile(string fileName)
   {
      TextReader reader = new StreamReader(fileName);
      string line = reader.ReadLine();
      Console.WriteLine(line);
      reader.Close();
   }
}
```

This program will compile successfully but if you run it, the result will look like the following (**FileNotFoundException** dumped on the console):

C:\Windows\system32\cmd.exe

```
Unhandled Exception: System.IO.FileNotFoundException: Could
not find file 'C:\Projects\OutOfRange\OutOfRange\bin\Debug\W
rongTextFile.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String m
aybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode,
FileAccess access, Int32 rights, Boolean useRights, FileShar
e share, Int32 bufferSize, FileOptions options, SECURITY_ATT
RIBUTES secAttrs, String msgPath, Boolean bFromProxy, Boolea
n useLongPath, Boolean checkHost)
   at System.IO.FileStream..ctor(String path, FileMode mode,
 FileAccess access, FileShare share, Int32 bufferSize, FileO
ptions options, String msgPath, Boolean bFromProxy, Boolean
useLongPath, Boolean checkHost)
   at System.IO.StreamReader..ctor(String path, Encoding enc
oding, Boolean detectEncodingFromByteOrderMarks, Int32 buffe
rSize, Boolean checkHost)
   at System.IO.StreamReader..ctor(String path)
   at ExceptionsDemo.ReadFile(String filename) in c:\Project
s\OutOfRange\OutOfRange\ExceptionsDemo.cs:line 14
   at ExceptionsDemo.Main() in c:\Projects\OutOfRange\OutOfR
ange\ExceptionsDemo.cs:line 9
Press any key to continue . . .
```

In this example, we have a code trying to open a text file for reading and then display the first line of this file on the screen. We will discuss working with files in more details in the chapter "Text Files".

The first two lines of **ReadFile()** contain code that throws an exception. In this example, if the file **WrongTextFile.txt** doesn't exist, the constructor **StreamReader(string, fileName)** will throw a **FileNotFoundException**. If an unexpected problem occurs during the input-output operations, the stream methods, such as **ReadLine()** will throw an **IOException**.

The code above will successfully compile but at run-time it will throw an exception if the **WrongTextFile.txt** file does not exist. The end result in this case is an error message displayed on the console. The console output also contains information of where and how the error occurred.

How Do Exceptions Work?

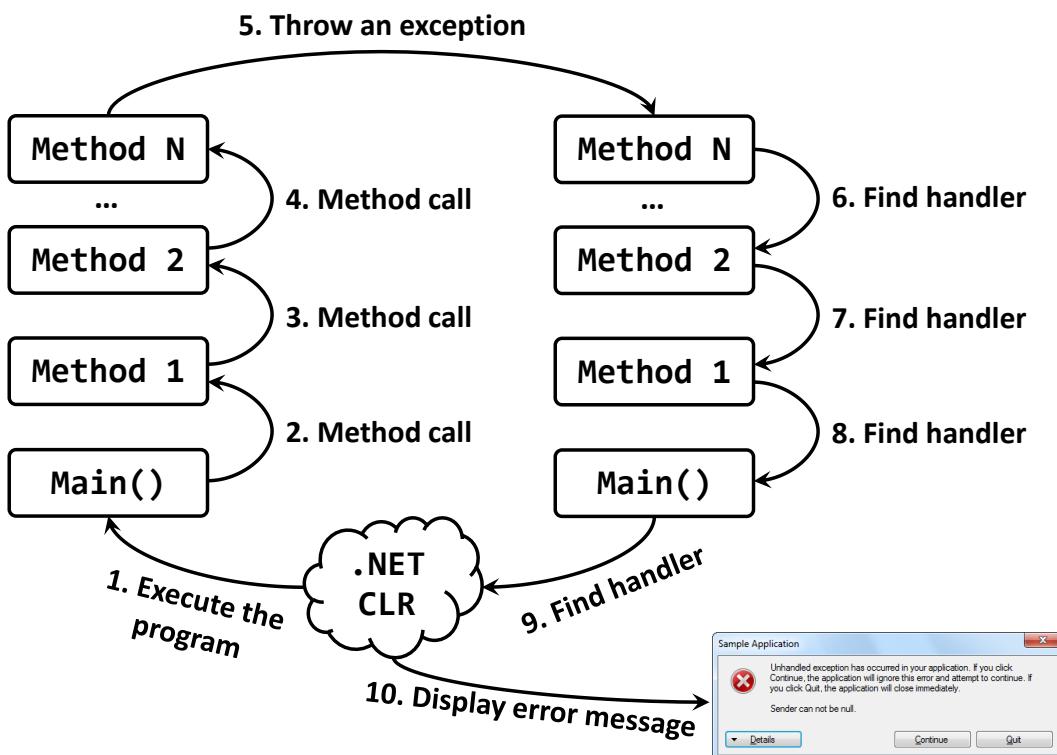
If during the normal program execution one of the methods throws an exception, the **normal flow of the program is interrupted**. In the example above this happens when the **StreamReader** is initialized. Let's take a look on the following line:

```
TextReader reader = new StreamReader("WrongTextFile.txt");
```

If this line triggers an error, the reader local variable will not be initialized and it will have its default value of **null**. None of the lines that follow in the method will be executed. The program will be interrupted until the CLR finds a handler that can process the exception.

Catching Exceptions in C#

After a method throws an exception, CLR is looking for an **exception handler** that can process the error. To understand how this works, we will take a closer look on the concept of a call-stack. The program **call-stack** is a stack structure that holds information about method calls, their local variables, method parameters and the memory for value types.

.NET programs start from the **Main(…)** method, which is the entry point of the program. Another method, let's name it "Method 1" could be called from **Main**. Let "Method 1" call "Method 2" and so on until "Method N" is called.

When "Method N" finishes, the program flow returns back to its calling method (in our example it would be "Method N-1"), then back to its calling method and so on. This goes on until the **Main(…)** method is reached. Once **Main(…)** finishes, the entire program exits.

The general principle is that when a new method is called, it is **pushed on top of the stack**. When the method finishes, it is **pulled back from the stack**. At any given point in time, the call-stack contains all the methods called during the execution – from the starting method **Main(…)** to the last

called method, which is currently executing, along with their local variables and arguments taken as input.

The **exception handling mechanism** follows a reversed process. When an exception is thrown, CLR begins searching an **exception handler** in the call-stack starting from the method that has thrown the exception. This is repeated for each of the methods down the call-stack until a handler is found which catches the exception. If **`Main(…)`** is reached and no handler is found, CLR catches the exception and usually displays an error message (either in the console or in a special error dialog box).

The described **method call and exception handling process** could be visualized in the following diagram (steps 1 through 5):

The try-catch Programming Construct

To handle an exception, we must surround the code that could throw an exception with a **`try-catch`** block:

```
try
{
   // Some code that may throw an exception
}
catch (ExceptionType objectName)
```

```
{
  // Code handling an Exception
}
catch (ExceptionType objectName)
{
  // Code handling an Exception
}
```

The **try-catch** construct consists of one **try** block and one or more **catch** blocks. Within the try block we put the code that could throw exceptions. The **ExceptionType** in the **catch** block must be a type, derived from **System.Exception** or the code wouldn't compile. The expression within brackets after **catch** is also a declaration of a variable, thus inside the **catch** block we can use **objectName** to use the properties of the exception or call its methods.

Catching Exceptions – Example

Let's now modify the code in our previous example to make it handle its exceptions. To do this, we wrap the code that could create problems in **try-catch** and then we add catch blocks to handle the two types of exceptions we know could arise.

```
static void ReadFile(string fileName)
{
  // Exceptions could be thrown in the code below
  try
  {
    TextReader reader = new StreamReader(fileName);
    string line = reader.ReadLine();
    Console.WriteLine(line);
    reader.Close();
  }
  catch (FileNotFoundException fnfe)
  {
    // Exception handler for FileNotFoundException
    // We just inform the user that there is no such file
    Console.WriteLine(
      "The file '{0}' is not found.", fileName);
  }
  catch (IOException ioe)
  {
    // Exception handler for other input/output exceptions
    // We just print the stack trace on the console
    Console.WriteLine(ioe.StackTrace);
```

```
    }
}
```

Now our method works in a different way. When **FileNotFoundException** is thrown during the **StreamReader** initialization when executing the constructor **new StreamReader(filename)**, the CLR will not execute the following lines but will jump to the row where we catch the exception **catch (FileNotFoundException fnfe)**:

```
catch (FileNotFoundException fnfe)
{
    // Exception handler for FileNotFoundException
    // We just inform the user that there is no such file
    Console.WriteLine("The file '{0}' is not found.", fileName);
}
```

In our example, users will simply be informed that such file does not exist by a message printed on the standard output:

```
The file 'WrongTextFile.txt' is not found.
```

Similarly, if an **IOException** is thrown during **reader.ReadLine()**, it is handled by the block below:

```
catch (IOException ioe)
{
    // Exception handler for FileNotFoundException
    // We just print the stack trace on the screen
    Console.WriteLine(ioe.StackTrace);
}
```

In this case, we display the exception stack trace on the standard output.

The lines between where the exception is thrown and the catch block that processed it are not executed.

Showing the full information about the exception to the end user is not always a good practice!

We will discuss the best practices in exception handling later in this chapter.

Stack Trace

The **stack trace** contains **detailed information about the exception** including where exactly it occurred in the program. The stack trace is very useful for programmers when they try to understand the problem causing the exception. The information in the stack trace is very technical and is designed

to be used by programmers and system administrators and not by the end users. During debugging the stack trace is a priceless tool.

Stack Trace – Example

Here is the stack trace from our first example:

```
Unhandled Exception: System.IO.FileNotFoundException: Could not
find file '…\WrongTextFile.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String
maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode,
FileAccess access, Int32 rights, Boolean useRights, FileShare
share, Int32 bufferSize, FileOptions options,
SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean
bFromProxy, Boolean useLongPath)
   at System.IO.FileStream..ctor(String path, FileMode mode,
FileAccess access, FileShare share, Int32 bufferSize,
FileOptions options)
   at System.IO.StreamReader..ctor(String path, Encoding
encoding, Boolean detectEncodingFromByteOrderMarks, Int32
bufferSize)
   at System.IO.StreamReader..ctor(String path)
   at Exceptions.Demo1.ReadFile(String fileName) in
Program.cs:line 17
   at Exceptions.Demo1.Main() in Program.cs:line 11
```

The system cannot find the file named "**WrongTextfile.txt**" and the **FileNotFoundException** is thrown.

Reading the Stack Trace

To be able to use the stack trace, we must be familiar with its structure.

The stack trace contains the following information:

- The full name of the exception class;
- A message with additional information about the error;
- Information about the call-stack;

In our example above, the full name of the exception is **System.IO.FileNotFoundException**. The error message follows: "**Could not find file '…\WrongTextFile.txt'.**" What follows is a full call-stack dump, which is usually the longest part of the stack trace. Each line of the call stack dump contains something similar to the following:

```
 at <namespace>.<class>.<method> in <source file>.cs:line <line>
```

Every method is shown in a separate line. On the first line is the method that threw the exception and on the least line – the **Main()** method (notice that the **Main()** method might not be present in case of an exception thrown by a thread which is not the main thread of the program). Every method is given with full information about the class that contains it and (if possible) even the line in the source code:

```
   at Exceptions.Demo1.ReadFile(String fileName) in
…\Program.cs:line 17
```

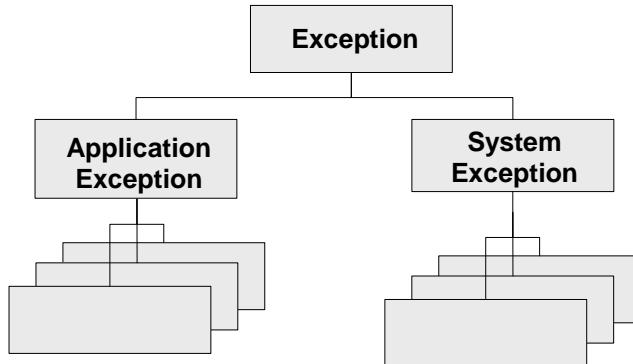
The line numbers are included only if the respective class **is compiled with debug information** (this information contains line numbers, variable names and other technical information). The debug information is not included in the .NET assemblies but is in separate files called 'debug symbols' (**.pdb**). As you can see in the example stack trace, debug information is available for some assemblies, while for others (like the .NET assemblies) it is not. This is why some entries in the stack trace have line numbers and others – not.

If the method throwing the exception is a constructor, then instead of method name, the stack trace contains the word **.ctor**, like in **System.IO.StreamReader..ctor(String path)**.

This rich information in the stack trace allows quickly and easily to find the class, the method and even the source line where the error has occurred. Then usually it is relatively straightforward to analyze the problem causing the error and fixing it. This is not the same in primitive languages such as C and Pascal where the concept of stack trace is not supported.

Throwing Exceptions (the throw Construct)

Exceptions in C# are thrown using the keyword **throw**. We need to provide an instance of the exception, containing all the necessary information about the error. Exceptions are normal classes and the only requirement is that they inherit directly or indirectly from the **System.Exception** class.

Here is an example:

```
static void Main()
{
   Exception e = new Exception("There was a problem");
   throw e;
}
```

The result from running this program is:

```
Unhandled Exception: System.Exception: There was a problem
   at Exceptions.Demo1.Main() in Program.cs:line 11
```

Exceptions Hierarchy

There are two types of exceptions in .NET Framework: exceptions thrown by the applications we develop (**ApplicationException**) and exceptions thrown by the runtime (**SystemException**). Each of these is a base class for a hierarchy of exception classes:

As all of these classes have different characteristics, we will examine them one by one.

The Exception Class

In .NET Framework, **Exception** is the base class for all exceptions. Several classes inherit directly from it, including **ApplicationException** and **SystemException**. These two classes are base classes for almost all exceptions that occur during the program execution.

The **Exception** class contains a copy of the call-stack at the time the exception instance was created. The class also has a (usually) short message describing the error (filled in by the method throwing the exception). Every exception could have a **nested exception** also sometimes called an **inner exception**, **wrapped exception** or **internal exception**.

The ability to wrap an exception with another exception is very useful in some cases and allows exceptions to be linked in the so called **exception chain**.

Exception – Constructors, Methods and Properties

Here is how the **System.Exception** class looks like:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
[ClassInterfaceAttribute(ClassInterfaceType.None)]
public class Exception : ISerializable, _Exception
{
  public Exception();
  public Exception(string message);
```

```
    public Exception(string message, Exception innerException);
    public virtual IDictionary Data { get; }
    public virtual string HelpLink { get; set; }
    protected int HResult { get; set; }
    public Exception InnerException { get; }
    public virtual string Message { get; }
    public virtual string Source { get; set; }
    public virtual string StackTrace { get; }
    public MethodBase TargetSite { get; }
    public virtual Exception GetBaseException();
}
```

The full specification of the **Exception** class given above is complex to be explained, so we will discuss only its most important methods and properties as they are inherited by all exceptions in .NET Framework.

- We have three constructors with different combinations for message and inner exception.
- The **Message** property returns a text description of the exception. For example if the exception is **FileNotFoundException**, the message could provide information which file was not found. In most of the cases, the code throwing the exception passes the message in the constructor. Once set, the **Message** property cannot be changed.
- The **InnerException** property returns the inner (wrapped, nested) exception or **null** if such doesn't exist.
- The **GetBaseException()** returns the innermost exception from a given exception chain. By definition, calling this method for every exception within an exception chain will always yield the same result – the first exception that happened.
- The **StackTrace** property returns information for the entire stack contained in the exception (we have already seen how this information looks like).

Application vs. System Exceptions

Exceptions in .NET are two types – system and application. System exceptions are defined in .NET libraries and are used by the framework, while application exceptions are defined by application developers and are used by the application software. When we, as developers, design our own exception classes, it is a good practice to inherit from **ApplicationException** and not directly from **SystemException** (or even worse – directly from **Exception**). **SystemException** should only be inherited internally within the .NET Framework.

Some of the worst system exceptions include **ExecutionEngineException** (which is thrown on internal error within CLR), **StackOverflowException** (call-stack overflow, most probably due to infinite recursion) and **OutOfMemoryException** (insufficient memory). In all of these cases, our application could hardly recover or react in some reasonable manner. Most frequently, when such exception occurs, the application just crashes.

Exceptions related to interaction with external components (like COM components) inherit from **ExternalException**. Examples are **COMException** and **Win32Exception**.

Throwing and Catching Exceptions

Let's look in more details at throwing and catching exceptions.

Nested Exceptions

We've already seen that each exception could contain a **nested** (**inner**) exception. Let's explain in more details why it is a common practice in OOP error handling to wrap exceptions in this way.

In software engineering, it is a good practice for every software component to define small number of specific **application exceptions**. The component then would throw only these specific application exceptions and not the standard .NET exceptions. In this way the users of the software component would know what exceptions could expect from it.

For instance, if we have a banking software and we have a component dealing with interests, this component would define (and throw) exceptions like **InterestCalculationException** and **InvalidPeriodException**. The interest component should not throw exceptions like **FileNotFoundException**, **DivideByZeroException** and **NullReferenceException**. When an error occurs, which is not directly related to interest calculation, the respective exception is wrapped in **InterestCalculationException** and the calling code will be informed that the interest calculation was not correctly done.

Still, these business application exceptions usually do not have detailed technical information about the nature of the problem. This is why, it is considered a good practice to include technical details about the problem and this is where inner exceptions come in handy. When the component throws its application exception, it should keep the original exception as an inner exception in order to preserve the technical details about the error.

Another example is when a software component (let's call it Component **A**) defines its own application exceptions (**A**-exceptions). This component internally uses another component (called Component **B**). If for some reason **B** throws a **B** exception (an exception defined in **B**), perhaps **A** will have to propagate the error because it will not be able to do its task. And because **A** cannot simply throw a B-exception, it must throw an **A**-exception, containing the **B**-exception as a nested exception.

There could be various reasons why **A** cannot simply throw a **B** exception:

- Component **A** users should not even know Component **B** exists (see the discussion regarding abstractions in the "Principles of OOP" chapter);
- Component **A** had not declared it would throw Component **B** exceptions;
- Component **A** users are not prepared to receive Component **B** exceptions. They expect component **A** exceptions only.

How to Read the Stack Trace with Nested Exceptions?

Below we have an example that creates an exception chain. We will demonstrate how such exception chain is created and how the stack trace looks like in the output:

```
37 static void Main()
38 {
39   try
40   {
41     string fileName = "WrongFileName.txt";
42     ReadFile(fileName);
43   }
44   catch (Exception e)
45   {
46     throw new ApplicationException("Smth. bad happened", e);
47   }
48 }
49 static void ReadFile(string fileName)
50 {
51   TextReader reader = new StreamReader(fileName);
52   string line = reader.ReadLine();
53   Console.WriteLine(line);
54   reader.Close();
55 }
```

In this example, we call the **ReadFile()** method (line 42), which will throw an exception (line 51) because the file "**WrongFileName.txt**" does not exist. In the **Main()** method we catch all exceptions (line 44), wrap them into a new exception of type **ApplicationException** and throw them again (line 46). As we shall see later in the section "Grouping Different Error Types", caching an **Exception** also catches all its descendant exceptions in its hierarchy. Finally the thrown exception (at line 46) is caught by .NET Framework and its stack trace is dumped on the console.

The result of running the above example is shown below:

C:\Windows\system32\cmd.exe

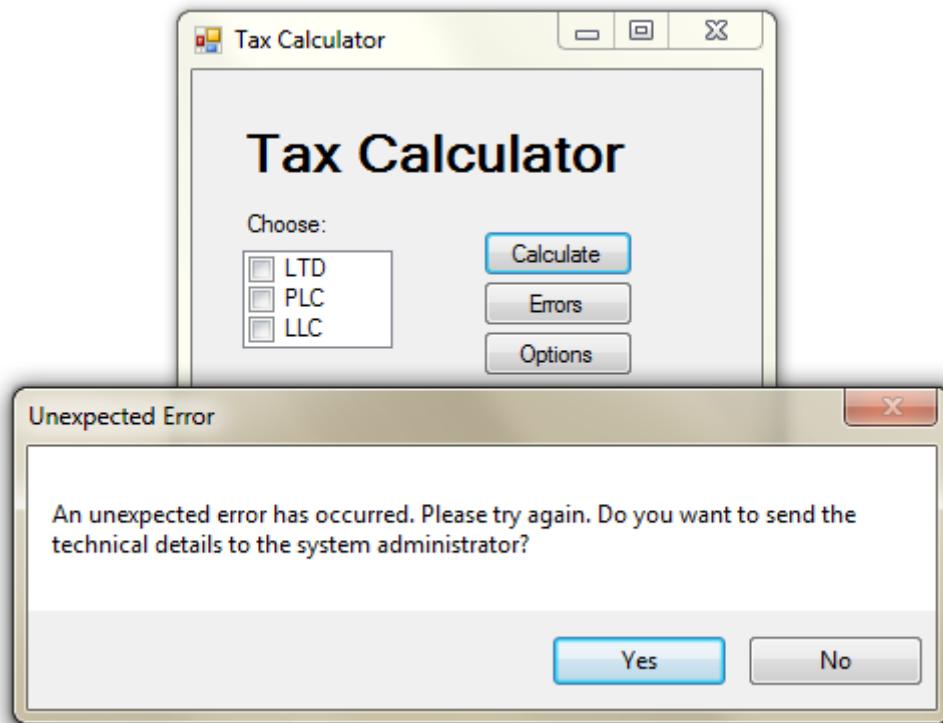
```
Unhandled Exception: System.ApplicationException: Smth. bad
happened ---> System.IO.FileNotFoundException: Could not fin
d file 'C:\Projects\OutOfRange\OutOfRange\bin\Debug\WrongFil
eName.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String m
aybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode,
FileAccess access, Int32 rights, Boolean useRights, FileShar
e share, Int32 bufferSize, FileOptions options, SECURITY_ATT
RIBUTES secAttrs, String msgPath, Boolean bFromProxy, Boolea
n useLongPath, Boolean checkHost)
   at System.IO.FileStream..ctor(String path, FileMode mode,
 FileAccess access, FileShare share, Int32 bufferSize, FileO
ptions options, String msgPath, Boolean bFromProxy, Boolean
useLongPath, Boolean checkHost)
   at System.IO.StreamReader..ctor(String path, Encoding enc
oding, Boolean detectEncodingFromByteOrderMarks, Int32 buffe
rSize, Boolean checkHost)
   at System.IO.StreamReader..ctor(String path)
   at ExceptionsDemo.ReadFile(String fileName) in c:\Project
s\OutOfRange\OutOfRange\ExceptionsDemo.cs:line 52
   at ExceptionsDemo.Main() in c:\Projects\OutOfRange\OutOfR
ange\ExceptionsDemo.cs:line 42
   --- End of inner exception stack trace ---
   at ExceptionsDemo.Main() in c:\Projects\OutOfRange\OutOfR
ange\ExceptionsDemo.cs:line 46
Press any key to continue . . .
```

Let's look more carefully at the stack trace. We now see an additional section marking the end of the nested exception:

```
--- End of inner exception stack trace ---
```

This gives useful information about how the exception was thrown.

If you look more closely on the first line, you will notice it contains information in the following format:

```
Unhandled Exception: Exception1: Msg1 ---> Exception2: Msg2
```

This shows that an exception of type **Exception1** is wrapped around an exception of type **Exception2**. After each exception type, we can see the message of the respective exception (as contained in the **Message** property). Using the information in the stack-trace (the file name, the method and the line number), we can find out how the exceptions occurred and where.

Visualizing Exceptions

In **console applications** errors are usually printed in the output although this might not be the most user-friendly way to notify the user for problems.

In **Web applications**, errors are frequently shown in the beginning or at the bottom of the page or near the UI field related to the error.

In **GUI applications** we should show the errors in a dialog window containing user-friendly description of the error. An example of user-friendly error message dialog box is given below:

As you can see, there is no single 'right' way to handle and visualize exceptions as it depends on the type of the application and its intended audience. Still there are some recommendations regarding how to handle exceptions and what is the best way to show them to the users. We will discuss these recommendations in the "Best Practices" section.

Which Exceptions to Handle and Which Not?

There is one universal rule regarding exception handling:

	A method should only handle exceptions which it expects and which it knows how to process. All the other exceptions must be left to the calling method.

If we follow this rule and every method leaves the exceptions it is not competent to process to the calling method, eventually we would reach the **`Main()`** method (or the starting method of the respective thread of execution) and if this method does not catch the exception, the CLR will display the error

on the console (or visualize it in some other way) and will terminate the program.

A method is competent to handle an exception if it expects this exception, it has the information why the exception has been thrown and what to do in this situation. If we have a method that must read a text file and return its contents as a string, that method might catch **FileNotFoundException** and return an empty string in this case. Still, this same method will hardly be able to correctly handle **OutOfMemoryException**. What should the method do in case of insufficient memory? Return an empty string? Throw some other exception? Do something completely different? So apparently the method is not competent to handle such exception and thus the best way is to pass the exception up to the calling method so it could (hopefully) be handled at some other level by a method competent to do it. Using this simple philosophy allows exception handling to be done in a structured and systematic way.

Throwing Exceptions from the Main() Method – Example

Throwing exceptions from the **Main()** method is generally not a good practice. Instead, it is better all exceptions to be caught in **Main()**. Still it is of course possible to throw exceptions from **Main()** just as from any other method:

```
static void Main()
{
   throw new Exception("Ooops!");
}
```

Every exception which is not handled in **Main()** is eventually caught by the CLR and visualized by printing the stack trace on the console output or in some other way. While for small applications it is not such a problem, big and complex applications generally should not crash in such ungraceful manner.

Catching Exceptions at Different Levels – Example

The ability to pass (or bubble) exceptions through a given method up to the calling method allows structured exception handling to be done at multiple levels. This means that we can catch certain types of exceptions in given methods and pass all other exceptions to the previous levels in the call-stack. In the example below, the exceptions in the **ReadFile()** method are handled at two levels (in the **try-catch** block of the **ReadFile()** method itself and in the **try-catch** block of the **Main()** method):

```
static void Main()
{
   try
```

```
  {
    string fileName = "WrongFileName.txt";
    ReadFile(fileName);
  }
  catch (Exception e)
  {
    throw new ApplicationException("Bad thing happened", e);
  }
}

static void ReadFile(string fileName)
{
  try
  {
    TextReader reader = new StreamReader(fileName);
    string line = reader.ReadLine();
    Console.WriteLine(line);
    reader.Close();
  }
  catch (FileNotFoundException fnfe)
  {
    Console.WriteLine("The file {0} does not exist!",
      filename);
  }
}
```

In this example the **ReadFile()** method catches and handles only **FileNotFoundException** while passing all other exceptions up to the **Main()** method. In the **Main()** method we handle only exceptions of type **IOException** and will let the CLR to handle all other exceptions (for instance, if **OutOfMemoryException** is thrown during program's execution, it will be handled by the CLR).

If the **Main()** method passes a wrong filename, **FileNotFoundException** will be thrown while initializing the **TextReader** in **ReadFile()**. This exception will be handled by the **ReadFile()** method itself. If on the other hand the file exists but there is some problem reading it (insufficient permissions, damaged file contents etc.), the respective exception that will be thrown will be handled in the **Main()** method.

Handling exceptions at different levels allows the error conditions to be handled at the most suitable place for the particular error. This allows the program code to be clear and structured and the flexibility achieved is enormous.

The try-finally Construct

Every **try** block could contain a respective **finally** block. The code within the **finally** block is always executed, no matter how the program flow leaves the try block. This guarantees that the **finally** block will be executed even if an exception is thrown or a return statement is executed within the **try** block.

The code in the finally block will not be executed if while executing the try block, CLR is unexpectedly terminated, e.g. if we stop the program through Windows Task Manager.

The basic form of the **finally** block is given below:

```
try
{
   // Some code that could or could not cause an exception
}
finally
{
   // Code here will always execute
}
```

Every **try** block may have zero or more **catch** blocks and at most one **finally** block. It is possible to have multiple **catch** blocks and a **finally** block in the same **try-catch-finally** construct.

```
try
{
   some code
}
catch (…)
{
   // Code handling an exception
}
catch (…)
{
   // Code handling another exception
}
finally
{
   // This code will always execute
}
```

When Should We Use try-finally?

In many applications we have to work with external for our programs resources. Examples for **external resources** include files, network connections, graphical elements, pipes and streams to or from different hardware devices (like printers, card readers and others). When we deal with such external resources, it is critically important to **free up the resources** as early as possible when the resource is no longer needed. For example, when we open a file to read its contents (let's say to load a JPG image), we must close the file right after we have read the contents. If we leave the file open, the operating system will prevent other users and applications to make certain operations on the file. Perhaps you faced such a situation when you could not delete some directory or a file because it is being used by a running process.

The **finally** block is priceless when we need to free an external resource or make any other cleanup. The **finally** block guarantees that the cleanup operations will not be accidentally skipped because of an unexpected exception or because of execution of **return**, **continue** or **break**.

Because proper resource management is an important concept in programming, we will look at it in some more details.

Resource Cleanup – Defining the Problem

In our example, we want to **read a file**. To accomplish this, we have a reader that must be closed when the file has been read. The best way to do this is to surround the lines using the reader in a **try-finally** block. Here is a refresh of how our example looks like:

```
static void ReadFile(string fileName)
{
  TextReader reader = new StreamReader(fileName);
  string line = reader.ReadLine();
  Console.WriteLine(line);
  reader.Close();
}
```

What is **the problem with this code**? Well, what the code is supposed to do is to open up a file reader, read the data and then close the reader before the method returns. This last part is a problem because the method could finish in one of several ways:

- An exception could be thrown when the reader is initialized (say if the file is missing).
- During reading the file, an exception could arise (imagine a file on a remote network device which goes offline during file reading).

- A **return** statement could be executed before the reader is closed (in our trivial example this would be obvious but it is not always as apparent).
- Everything goes as expected and the method is executed normally.

So our method as written in the example above has a **critical flaw**: it will close the reader only in the last scenario. In all of the other cases, the code closing the reader will not be executed. And if this code is within a loop, things get even more complex as **continue** and **break** operators must be considered too.

Resource Cleanup – Solving the Problem

In the previous section we explained the fundamental flaw of the solution '**open the file → read → close**'. If an error occurs during opening or reading the file, we will leave the file open.

To solve this, we can use the **try-finally** construct. We will first discuss the case in which we have one resource to clean-up (in this case a file). Then we will give an example when we have two or more resources.

Closing a file stream could be done using the following pattern:

```
static void ReadFile(string fileName)
{
   TextReader reader = null;
   try
   {
      reader = new StreamReader(fileName);
      string line = reader.ReadLine();
      Console.WriteLine(line);
   }
   finally
   {
      // Always close "reader" (if it was opened)
      if (reader != null)
      {
         reader.Close();
      }
   }
}
```

In this example we first declare the **reader** variable, and then initialize the **TextReader** in a **try** block. Then in the **finally** block we close the reader. Whatever happens during **TextReader**'s initialization or during reading, it is guaranteed that the file will be closed. If there is a problem initializing the reader (say the file is missing), then **reader** will remain **null** and this is why we do a check for **null** in the finally block before calling **Close()**. If the value

is indeed **null**, then the reader has not been initialized and there is no need to close it. The code above guarantees that if the file has been opened, then it will be closed no matter how the method exits.

The example above should, in principle, properly handle all exceptions related to opening and initialization of the reader (like **FileNotFoundException**). In our example, these exceptions are not handled and are simply propagated to the caller.

We have chosen file streams for our example for freeing resources up but the same principle applies to all resources that require proper cleanup. These could be remote connections, operating system resources, database connections and so on.

Resource Cleanup – Better Solution

While the above solution is correct, it is unnecessary complex. Let's look at a **simplified version**:

```
static void ReadFile(string fileName)
{
  TextReader reader = new StreamReader(fileName);
  try
  {
    string line = reader.ReadLine();
    Console.WriteLine(line);
  }
  finally
  {
    reader.Close();
  }
}
```

This code has the advantage of being **simpler and shorter**. We avoid the preliminary declaration of the **reader** variable and the check for **null** in the finally block. The **null** check is now not necessary because the initialization of the reader is outside of the **try** block and if an exception occurs during the initialization, the **finally** block will not be executed at all.

This code is cleaner, shorter and clearer and is known as "**dispose pattern**". However, note that this way the exception will go up to the method calling **ReadFile(…)**.

Multiple Resources Cleanup

Sometimes we need to free more than one resource. It is a good practice to free the resources in in reverse order in respect to their allocation.

We can use the same approach outlined above, nesting the **try-finally** blocks inside each other:

```
static void ReadFile(string filename)
{
   Resource r1 = new Resource1();
   try
   {
      Resource r2 = new Resource2();
      try
      {
         // Use r1 and r2
      }
      finally
      {
         r2.Release();
      }
   }
   finally
   {
      r1.Release();
   }
}
```

Another option is to declare all of the resources in advance and then make the cleanup in a single **finally** block with respective **null** checks:

```
static void ReadFile(string filename)
{
   Resource r1 = null;
   Resource r2 = null;
   try
   {
      Resource r1 = new Resource1();
      Resource r2 = new Resource2();

      // Use r1 and r2
   }
   finally
   {
      if (r1 != null)
      {
         r1.Release();
      }
      if (r2 != null)
```

```
        {
            r2.Release();
        }
    }
}
```

Both of these options are **correct** and both are applied depending on the situation and programmer's preference. The second approach is a little bit riskier as if an exception occurs in the **finally** block, some of the resources will not be cleaned up. In the example above, if an exception is thrown during **r1.Release()**, **r2** will not be cleaned up. If we use the first option, there is no such problem but the code is a bit longer.

IDisposable and the "using" Statement

It is time to present a new shorter and simplified way to release some kinds of resources in C#. We will demonstrate which resources can use this special programming construct and how it looks like.

IDisposable

The main use of **IDisposable** interface is to **release resources**. In .NET such resources are window handles, files, streams and others. We will talk about interfaces in "OOP Principles" chapter. Now we may consider interface as an indication that given type of objects (for example streams for reading files) support a certain number of operations (for example closing the stream and releasing related resources).

We will not go into details how to implement **IDisposable** since we have to go much deeper and explain how the garbage collector works, how to use destructors, unmanaged resources and so on.

The important method in **IDisposable** interface is **Dispose()**. The main thing we need to know about the method is that it **releases the resources** of the class that implements it. In cases when resources are streams, readers or files releasing resources can be done using the **Dispose()** method from **IDisposable** interface, which calls their **Close()** method. This method closes them and releases their resources. So to close a stream we can do the following:

```
StreamReader reader = new StreamReader(fileName);
try
{
    // Use the reader here
}
finally
{
```

```
   if (reader != null)
   {
      reader.Dispose();
   }
}
```

The Keyword "using"

The previous example can be written in shorter form with the help of the **using keyword in C#**, as shown in the following example:

```
using (StreamReader reader = new StreamReader(fileName))
{
   // Use the reader here
}
```

The above **simplified form of the "dispose pattern"** is simple to write, simple to use and simple to read and is guaranteed to release correctly the allocated resources specified in the brackets of the **using** statement.

It is not necessary to have **try-finally** or to explicitly call any method to release the resources. The compiler takes care to automatically put **try-finally** block and the used resources are released by calling the **Dispose()** method after leaving the **using** block.

Later in chapter "Text Files" we will extensively use the **using** statement to correctly read and write text files.

Nested "using" Statements

The **using** statements can be nested one within another:

```
using (ResourceType r1 = …)
   using (ResourceType r2 = …)
      …
         using (ResourceType rN = …)
            statements;
```

The previous example can be written like this:

```
using (ResourceType r1 = …, r2 = …, …, rN = …)
{
   statements;
}
```

It is important to mention that **using** statement is not related to exception handling. Its only purpose is to release the resources no matter whether exceptions are thrown or not. It does not handle exception.

When to Use the "using" Statement?

There is a simple rule when to use **using** with .NET classes:

Use the using statement with all classes that implement the IDisposable interface. Look for IDisposable in MSDN.

When a class implements **IDisposable** interface this means that the creator of this class expects it can be **used with the using statement** and the class contains some expensive resource that should not be left unreleased. Implementing **IDisposable** also means that it should be released immediately after we finish using the class and the easiest way to do this in C# is with **using** statement.

Advantages of Using Exceptions

So far we reviewed the exceptions in details, their characteristics and how to use them. Now let's find out why they were introduced and why they are so widely used.

Separation of the Exception Handling Code

Using exceptions allow us to separate the code, which describes the normal execution of the program from the code required for unexpected execution and the code for error handling. We will demonstrate this **separation concept** in the following example:

```
void ReadFile()
{
  OpenTheFile();
  while (FileHasMoreLines)
  {
    ReadNextLineFromTheFile();
    PrintTheLine();
  }
  CloseTheFile();
}
```

Let's explore the example step by step. It does the following:

- Open the file;
- While the file has more lines:
    - Read the next line from the file;

- Print the line;
- Close the file;

The method looks good but a closer look brings up some questions:

- What will happen if the file does not exist?
- What will happen if the file cannot be opened?
- What will happen if reading a line fails?
- What will happen if the file cannot be closed?

Error Handling without Exceptions

Let's change the method having these questions in mind **without using exceptions**. Let's **use error codes** returned by any method that we use. Using error codes is standard way for handling errors in procedure oriented programming, where every method returns **int**, which provides information whether the method was executed correctly. Error code 0 means that everything is correct. Any other code means some error. Different kinds of errors have different codes (usually it is a negative number).

```
int ReadFile()
{
  errorCode = 0;
  openFileErrorCode = OpenTheFile();

  // Check whether the file is open
  if (openFileErrorCode == 0)
  {
    while (FileHasMoreLines)
    {
      readLineErrorCode = ReadNextLineFromTheFile();
      if (readLineErrorCode == 0)
      {
        // Line has been read properly
        PrintTheLine();
      }
      else
      {
        // Error during line reading
        errorCode = -1;
        break;
      }
    }
    closeFileErrorCode = CloseTheFile();
    if (closeFileErrorCode != 0 && errorCode == 0)
```

```
    {
      errorCode = -2;
    }
    else
    {
      errorCode = -3;
    }
  }
  else if (openFileErrorCode == -1)
  {
    // File does not exist
    errorCode = -4;
  }
  else if (openFileErrorCode == -2)
  {
    // File can't be open
    errorCode = -5;
  }
  return errorCode;
}
```

As a result we have a **hard to understand and easy to break "spaghetti" code**. Program logic is mixed with the error handling logic. Big parts of the code are the rules for error handling. Errors don't have type, description or stack trace and we have to wonder what the different error codes mean.

Error Handling with Exceptions

We can avoid all of the above spaghetti code just by **using exceptions**. Here is how the same method will look like using exceptions instead:

```
void ReadFile()
{
  try
  {
    OpenTheFile();
    while (FileHasMoreLines)
    {
      ReadNextLineFromTheFile();
      PrintTheLine();
    }
  }
  catch (FileNotFoundException)
  {
    DoSomething();
```

```
  }
  catch (IOException)
  {
    DoSomethingElse();
  }
  finally
  {
    CloseTheFile();
  }
}
```

In fact exceptions don't save us the effort in finding and processing errors but give us **more elegant, short, clear and efficient way** to do it.

Grouping Different Error Types

The hierarchical nature of exceptions allows us to catch and handle whole groups of exceptions at one time. When using **catch** we are not only catching the given type of exception but **the whole hierarchy of exception types that are inheritors of the declared type**.

```
catch (IOException e)
{
  // Handle IOException and all its descendants
}
```

The example above will catch not only the **IOException**, but all of its descendants including **FileNotFoundException**, **EndOfStreamException**, **PathTooLongException** and many others. In the same time exceptions like **UnauthorizedAccessException** and **OutOfMemoryException** will not be caught, because they don't inherit from **IOException**. We can look in MSDN for the exceptions hierarchy if we wander which exceptions to catch.

It is not a good practice, but it is possible to catch all exceptions:

```
catch (Exception e)
{
  // A (too) general exception handler
}
```

Catching **Exception** and all of its inheritors is not a good practice. It is better to catch more specific groups of exceptions like **IOException** or just one type of exception like for example **FileNotFoundException**.

Catching Exceptions at the Most Appropriate Place

The ability to **catch exceptions at multiple locations** is extremely comfortable. It allows us to **handle the exception at the most appropriate place**. Let's demonstrate this with a simple comparison with the old approach using error codes. Let's have the following method structure:

```
Method3()
{
  Method2();
}

Method2()
{
  Method1();
}

Method1()
{
  ReadFile();
}
```

The method **Method3()** calls **Method2()**, which calls **Method1()** where **ReadFile()** is called. Let's suppose that **Method3()** is the method interested in eventual error in the **ReadFile()** method. If such error occurs in **ReadFile()** it wouldn't be easy to transfer the error to **Method3()** using the **traditional approach with error codes**:

```
void Method3()
{
  errorCode = Method2();
  if (errorCode != 0)
    process the error;
  else
    DoTheActualWork();
}

int Method2()
{
  errorCode = Method1();
  if (errorCode != 0)
    return errorCode;
  else
    DoTheActualWork();
}
```

```
int Method1()
{
   errorCode = ReadFile();
   if (errorCode != 0)
      return errorCode;
   else
      DoTheActualWork();
}
```

First in **Method1()** we have to analyze the error code returned by **ReadFile()** method and eventually pass it to **Method2()**. In **Method2()** we have to analyze the error code returned by **Method1()** and eventually pass it to **Method3()** where to handle the error itself.

How can we avoid all this? Let's remember that that the CLR searches for exceptions back in the call stack of the methods and lets each of them to define catching and handling of the exceptions. If the method is not interested in catching some exception it is simply sent back in the stack:

```
void Method3()
{
   try
   {
      Method2();
   }
   catch (Exception e)
   {
      process the exception;
   }
}

void Method2()
{
   Method1();
}

void Method1()
{
   ReadFile();
}
```

If an error occurs during reading the file it will be ignored in **Method1()** and **Method2()** and will be caught and handled in **Method3()** where is the most appropriate place to handle the error. Let's remember again the most

important rule: every method should catch only exceptions that can handle and skip all the others.

Best Practices when Using Exceptions

In this section we will give some recommendations and **best practices for correctly using exceptions** for error handling and unexpected situations. These are important rules that should be remembered and followed.

When to Rely on Exceptions?

To understand when it is good to rely on exceptions let's see the following example: we have a program that opens a file by given path and file name. While writing the user can write the file name wrong. This should rather be considered normal and not exceptional.

We can be prepared and first check if the file exists before we try to open it:

```
static void ReadFile(string fileName)
{
  if (!File.Exists(fileName))
  {
    Console.WriteLine(
      "The file '{0}' does not exist.", fileName);
    return;
  }

  StreamReader reader = new StreamReader(fileName);
  using (reader)
  {
    while (!reader.EndOfStream)
    {
      string line = reader.ReadLine();
      Console.WriteLine(line);
    }
  }
}
```

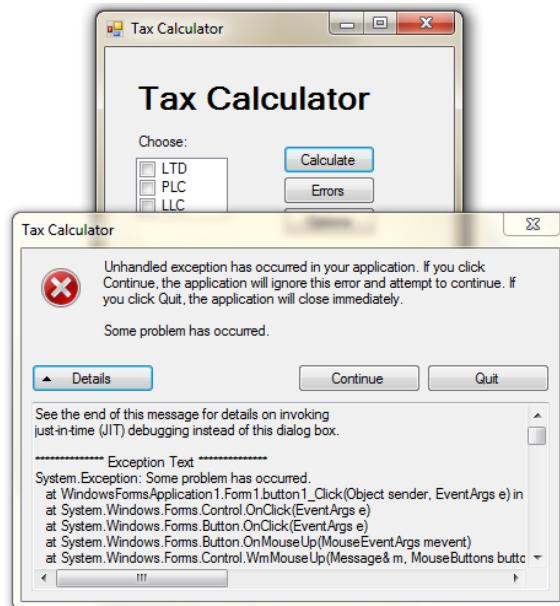
If we call the method and the file is missing we will see the following message in the console:

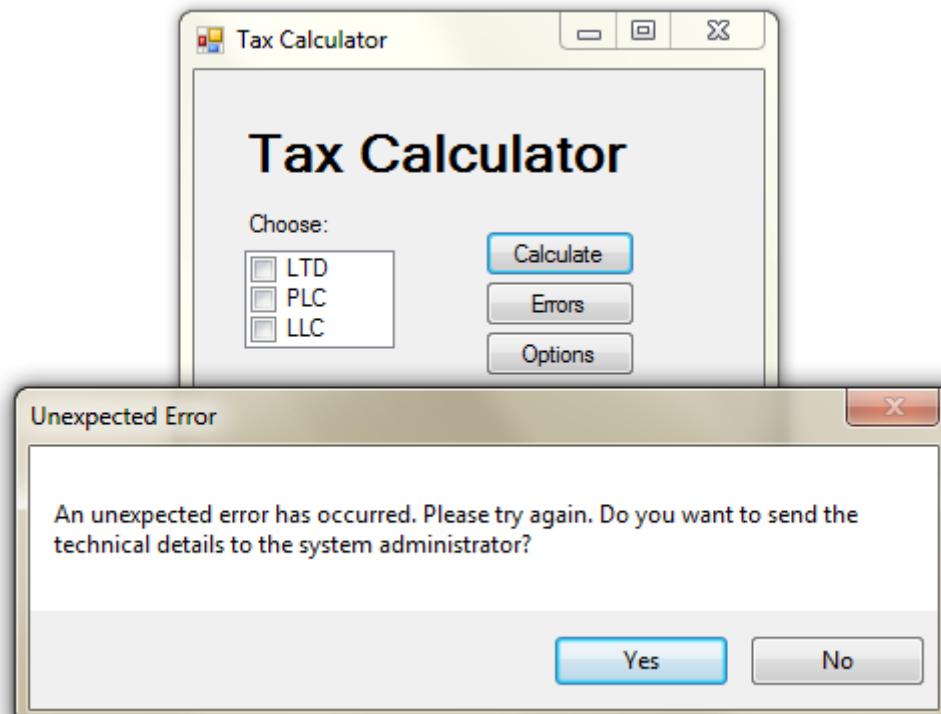
```
The file 'WrongTextFile.txt' does not exist.
```

The other way to implement this is the following:

```
static void ReadFile(string filename)
{
```

```
    StreamReader reader = null;
    try
    {
        reader = new StreamReader(filename);
        while (!reader.EndOfStream)
        {
            string line = reader.ReadLine();
            Console.WriteLine(line);
        }
        reader.Close();
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine(
            "The file '{0}' does not exist.", filename);
    }
    finally
    {
        if (reader != null)
        {
            reader.Close();
        }
    }
}
```

We can consider the second option as worse because exceptions should be used for unexpected situations and missing file is more or less usual.

It is not a good practice to rely on exceptions for expected events for another reason: performance. Throwing an exception is **time consuming operation**. An object has to be created to hold the exception, the stack trace has to be initialized and handler for this exception has to be found and so on.

It is hard to define the exact border between expected and unexpected. In general expected event is something related to the program functionality. Input of wrong file name for example. Power cut during the execution of the program, from the other hand, is unexpected event.

Throw Exceptions to the End User?

Exceptions are **confusing for most users**. They give the impression of a poorly written program that "has bugs". What will the user of our application entering invoices think if suddenly the program shows this dialogue?

This dialogue is very suitable for a developers or administrators for example, but it is extremely inappropriate for the end users.

Instead of this dialogue we can show another one, much more user friendly and understandable for the user:

This is the **good way to show the error message to the end user**. The message is easy to understand from the user and also contains technical details that can be used if required but is not visible at the beginning.

It is recommended when exceptions are not caught by anyone (such exceptions can only be runtime errors) to be caught by a **global exception handler** which saves them on the disk and shows user friendly message such as “An error occurred, please try again later”. It is a good a practice to show not only a user friendly message but also technical information (stack trace) available on demand (e.g. through an additional button or link).

Throw Exceptions at the Appropriate Level of Abstraction!

When we throw our own exceptions we must keep in mind the abstractions in the context our methods work. For example if our method works with arrays we can throw **IndexOutOfRangeException** or **NullReferenceException** because our method works at low level and directly operates with the memory and the array elements. But if our method is doing accumulating of interests at all accounts in a bank it should not throw **IndexOutOfRangeException** because this exception is not from the business area of the banking sector. It would be normal accumulation of interests in a bank software to throw **InvalidInterestException** exception with an appropriate error message where the original **IndexOutOfRangeException** exception to be attached.

Let’s give another example: we call a method that sorts an array of integers and throws an exception **TransactionAbortedException**. This is also an inappropriate exception just as **NullReferenceException** was in accumulation of interests in the bank software. That is why we should consider the abstraction level where our method works when we throw our exception.

If Your Exception Has a Source, Use It!

When we catch an exception and throw a new one with a higher level of abstraction we should **always attach the original exception** to it. This way the user of our code will be able to easily find the exact reason for the error and the location where it occurred at the first place.

This rule is a special case of more general rule:

Each exception should carry detailed information about the problem.

From the rule above many more rules come out: we should have a relevant error message, the error type should match the problem and the exceptions should hold its source as inner exception.

Give a Detailed Descriptive Error Message!

The **error message** that every exceptions holds is **extremely important**. In most cases it is enough to give us information what is the problem. If the message is not good enough the users of your methods will not be able to quickly solve the problem.

Let's take the following example: we have a method that reads the applications settings from a file. For example size and position of all windows in the application and others. There is a problem while reading the settings file and we receive the following error message:

```
Error.
```

Is this enough to find the problem? Obviously not. What should be the message so it is descriptive enough? Is this one better?

```
Error reading settings file.
```

Obviously the message above is better but it is still not good enough. It explains what the error is but does not tell us what causes it. Let's suppose we change the program so it gives the following error information:

```
Error reading settings file:
C:\Users\Administrator\MyApp\MyApp.settings
```

This error message is better because it tells us which file caused the problem (something that would save us time, especially if we are not familiar with the application and don't know where it keeps its settings files).

The situation could be even worse – we may not have the source code of the application and don't have the access to the stack trace (if we have compiled without debug information). That is why the error message should be even better. For example like the following:

```
Error reading settings file:
C:\Users\Administrator\MyApp\MyApp.settings. Number expected at
line 17.
```

This message fully describes the problem. Obviously we have an error on line 17, in **`MyApp.settings`** file, which is in **`C:\Users\Administrator\MyApp`** folder. On this line a number is expected but is not provided. If we open the file we could quickly find the problem.

Always give adequate, detailed and correct error message when throwing exceptions! The user of your code should be able to tell what and where is the problem and what caused it when reading the error message.

Let's give some examples:

- We have a method that searches for an integer in an array. If it throws **IndexOutOfRangeException** it is important to mention the index that cannot be reached in the error message. For example index 18 when the length of the array is 7. If we don't know the position we will hardly understand why we are outside the array.
- We have a method that reads integers from a file. If in the file we have a row without an integer we should get an error, which explains that at row 17 for example an integer is expected instead of a string (and prints the string).
- We have a method that calculates the sum of numeric expression. If we find an error in the expression the exception should say what error occurred and at what position. The code that causes the error may use **String.Format(…)** to build the error message. Here is an example how to implement this:

```
throw new FormatException(
  string.Format("Invalid character at position {0}. " +
  "Number expected but character '{1}' found.", index, ch));
```

Error Messages with Wrong Content

Even worse than throwing an exception with not enough information is throwing one with wrong information. If in the last example we say the error is at row 3 instead of row 17 this will be misleading and will be worse than just showing an error and give no details.

Be careful not to show messages with incorrect content!

Use English for All Exception Messages

Use **English for the error messages** when throwing an exception. This rule is a sub-rule of the rule "use **English** in your entire source code. The reason: English is the only language that is understood by programmers around the world. One day your code could be used by foreigners. If you live in France you probably won't be happy to get error messages in Chinese and vice-versa, would you?

Note that error messages shown to the end user could be in his native language, but the **error messages in the exceptions should always be in English**. The exceptions are for the developer. The developers around the world use English. The messages (errors / notifications / warnings) for the end user are different story. These messages could be in the language which is best suited for the end-users and may be customized through localization techniques like resources, embedded resource files and resource strings (see

http://msdn.microsoft.com/en-us/magazine/cc163609.aspx for additional information).

Never Ignore the Exceptions You Catch!

Never ignore the exceptions you catch without handling them. Here is an example what we should **not** do:

```
try
{
    string fileName = "WrongTextFile.txt";
    ReadFile(fileName);
}
catch (Exception e)
{ }
```

In the example the exception is caught and ignored. This means that if the file is missing the program will not read anything and there will not be any error message. This gives the user wrong impression the file is read when it is in fact missing. **Don't do this!**

If we ever need to ignore an exception on purpose we should add a comment, which will help us when reading the code later. Here is an example:

```
int number = 0;
try
{
    string line = Console.ReadLine();
    number = Int32.Parse(line);
}
catch (Exception)
{
    // Incorrect numbers are intentionally considered 0
}
Console.WriteLine("The number is: " + number);
```

We can improve the code above by using **Int32.TryParse(…)** or by initializing the **number** variable with 0 in the **catch** block, not outside of it. In the second case the comment in the code and empty **catch** block are not necessary.

Dump the Error Messages in Extreme Cases Only!

Let’s take our method, which is reading the application settings from a file. If an error occurs it could print it in the console but what will happen with the calling method? It will suppose that the settings are read correctly.

There is an important concept in programming:

A method should either do the work it is created for or throw an exception. Any other behavior is incorrect!

This is a very important rule that is why we will repeat it and even extend it:

A method should either do the work it is created for or throw an exception. In case of wrong input the method should throw an exception and should not return a wrong result!

We can explain the rule in details: A method is created to do a certain job. What the method is doing should be clear from its name. If we cannot give an appropriate name to the method means that it is doing many things and we should split it so everything is in separate method. If the method cannot do the work it is created for it should throw an exception. For example if we have a method for sorting of an array of integers. If the array is empty the method should either return an empty array or return an error. Wrong input should cause an exception and not return a wrong result! For example if we try to take a substring from index 7 to 12 from a string with length 10, it should cause an exception and not return fewer characters. This is how the **`Substring()`** method in **`String`** works.

We will give another example, which confirms the rule that a method should do the work it is created for or throw an exception. Let's suppose we copy a big file from the local disk to an USB flash drive. It could happen so that the space on the flash drive is not enough and the file cannot be copied. Which of the following is **correct** and the program for coping files (for example Windows Explorer) should do?

- The file is not copied and no error message is shown.
- The file is partially copied and no error message is shown.
- The file is partially copied and error message is shown.
- The file is not copied and error message is shown.

From the user point of view the only correct behavior of the program is the last one: if a problem occurs the file should not be copied partially and an error message should be shown. We should do the same if we have to write a method that copy files. It should fully copy the given file or throw an exception. At the same time it should not leave any traces – it should delete any partial result if such was created.

Don't Catch All Exceptions!

A very common mistake with exceptions is to catch all exceptions no matter what type they are. Here is an example where all exceptions are handled wrong:

```
try
```

```
{
  ReadFile("CorrectTextFile.txt");
}
catch (Exception)
{
  Console.WriteLine("File not found.");
}
```

In the code we suppose that there is a method **ReadFile()**, which reads a text file and returns the content as **string**. The **catch** block catches all exceptions (regardless of their type), not only **FileNotFoundException**, and in all cases prints that file is not found. There are unexpected situations such as when file is locked by another process in the operating system. In such case the CLR will generate **UnauthorizedAccessException**, but the message that the program will show to the user will be wrong and misleading. The file exists but the program will claim it is not there. The same will happen when during the file opening we are out of memory and **OurOfMemoryException** is generated. The message will be incorrect again.

Only Catch Exceptions You Know How to Process!

We should handle only errors that we expect and we are prepared for. We should leave the other errors (exceptions) so they are caught by another method that knows how to handle them.

A method should not catch all exceptions – it should only catch the ones it can process correctly.

This is a very important rule that should be followed. If you don't know how to handle an exception do not catch it or wrap it with your exception and pass it on for additional handling.

Exercises

1. Find out all exceptions in the **System.IO.IOException hierarchy**.

2. Find out all standard exceptions that are part of the **hierarchy** holding the class **System.IO.FileNotFoundException**.

3. Find out all standard exceptions from **System.ApplicationException hierarchy**.

4. Explain the concept of **exceptions** and **exception handling**, when they are used and how to **catch** exceptions.

5. Explain when the statement **try-finally** is used. Explain the relationship between the statements **try-finally** and **using**.

6. Explain the **advantages** when using exceptions.

7. Write a program that takes a positive integer from the console and prints the **square root** of this integer. If the input is **negative or invalid** print "Invalid Number" in the console. In all cases print "Good Bye".

8. Write a method **ReadNumber(int start, int end)** that reads an integer from the console in the range [**start…end**]. In case the input integer is not valid or it is not in the required range throw appropriate exception. Using this method, write a program that takes 10 integers $a_1, a_2, \ldots, a_{10}$ such that $1 < a_1 < \ldots < a_{10} < 100$.

9. Write a method that takes as a parameter the name of a **text file**, **reads the file and returns its content as string**. What should the method do if and **exception is thrown**?

10. Write a method that takes as a parameter the name of a binary file, **reads the content** of the file and returns it as an array of bytes. Write a method that **writes the file content** to another file. Compare both files.

11. Search for information in Internet and define your own class for exception **FileParseException**. The exception has to contain the name of the processed file and the number of the row where the problem is occurred. Add appropriate constructors in the exception. Write a program that reads integers from a text file. If the during reading a row does not contain an integer throw **FileParseException** and pass it to the calling method.

12. Write a program that gets from the user the full path to a file (for example **C:\Windows\win.ini**), reads the content of the file and prints it to the console. Find in MSDN how to us the **System.IO.File.ReadAllText(…)** method. Make sure all possible exceptions will be caught and a user-friendly message will be printed on the console.

13. Write a program to **download a file from Internet** by given URL, e.g. http://introprogramming.info/wp-content/themes/introprograming_en/images/Intro-Csharp-Book-front-cover-big_en.png.

Solutions and Guidelines

1. Search in MSDN. The easiest way to do this is to search in Google for "**IOException MSDN**" (without the quotes).

2. Look at the instructions for the **previous task**.

3. Look at the instructions for the **previous task**.

4. Use the information from the **section "What Is an Exception?"** earlier in this chapter.

5. When having difficulties use the information from the **section "try-finally Construct"**.

6. When having difficulties use the information from the **section "Exceptions Advantages"**.

7. Create **try-catch-finally** statement.

8. When invalid number is used we can throw **Exception** because there is no other exception that can better describe the problem. As an alternative we can define our own exception class called in a way that better describes the problem, e.g. **InvalidNumberException**.

9. First read the chapter "Text Files". Read the file line by line with **System.IO.StreamReader** class and add the rows in **System.Text.StringBuilder**. Throw all exceptions from the method without catching them. You may cheat and solve the problem in one line of code by using the static method **System.IO.File.ReadAllText()**.

10. It is not too likely to write this method correctly without external help. Search in Internet to learn more about **binary streams**. After that follow the instructions below for reading a file:

    - For reading use **FileStream** and write the data in a **MemoryStream**. You have to read the file in parts, for example on portions with 64 KB each, the last one can be smaller.

    - Be careful with the method for reading the bytes **FileStream.Read(byte[] buffer, int offset, int count)**. This method **can read less bytes than requested**. You have to write as many bytes as you read from the input stream. Create a loop that ends when zero bytes are read.

    - Use **using** to correctly closing the streams.

    Saving an array of bytes in a file is a simpler task. Open **FileStream** and start writing the bytes inside from the **MemoryStream**. Use **using** to correctly close the streams.

    Use a **big ZIP archive** to test (for example 300 MB). If the program is not working correctly it will break the structure of the archive and an error will occur when trying to open it.

    You can cheat by using the system methods **System.IO.File.ReadAllBytes()** and **System.IO.File.WriteAllBytes(byte[])**.

11. Inherit from **Exception** class and add a constructor to it. For example **FileParseException(string message, string filename, int line)**. Use this exception the same way as using any other exception. The number can be read with **StreamReader** class.

12. Search for all possible exceptions that the method could throw and for all of them define a **catch** block and print user-friendly message.

13. Search for articles in Internet for "**downloading a file with C#**" or search for information and examples about using the **WebClient** class. Make sure you catch and process all **exceptions** that can be thrown.

Chapter 13. Strings and Text Processing

In This Chapter

In this chapter we will explore **strings**. We are going to explain how they are implemented in C# and in what way we can process text content. Additionally, we will go through different methods for **manipulating a text**: we will learn how to compare strings, how to search for substrings, how to extract substrings upon previously settled parameters and last but not least how to split a string by separator chars. We will demonstrate how to **correctly build strings** with the **StringBuilder** class. We will provide a short but very useful information for the most commonly used **regular expressions**. We will discuss some classes for efficient construction of strings. Finally, we will take a look at the methods and classes for achieving more elegant and stricter **formatting** of the text content.

Strings

In practice we often come to the **text processing**: reading text files, searching for keywords and replacing them in a paragraph, validating user input data, etc… In such cases we can save the text content, which we will need in strings, and process them using the C# language.

What Is a String?

A string is a **sequence of characters** stored in a certain address in memory. Remember the type **char**? In the variable of type **char** we can record only one character. Where it is necessary to process more than one character then strings come to our aid.

In. NET Framework each character has a serial number from the **Unicode table**. The Unicode standard is established in the late 80s and early 90s in order to store different types of text data. Its predecessor **ASCII** is able to record only 128 or 256 characters (respective ASCII standard with 7-bit or 8-bit table). Unfortunately, this often does not meet user needs – as we can fit in 128 characters only digits, uppercase and lowercase Latin letters and some specific individual characters. When you have to work with text in Cyrillic or other specific language (e.g. Chinese or Arabian), 128 or 256 characters are extremely insufficient. Here is why .NET uses 16-bit code table for the characters. With our knowledge of number systems and representation of information in computers, we can calculate that the code table store 2^16 =

65,536 characters. Some characters are encoded in a specific way, so it is possible to use two characters of the Unicode table to create a new character – the resulting signs exceed 100,000.

The System.String Class

The class **System.String** enables us to handle **strings in C#**. For declaring the strings we will continue using the **keyword string**, which is an alias in C# of the **System.String** class from .NET Framework. The work with string facilitates us in manipulating the text content: construction of texts, text search and many other operations.

Example of **declaring a string**:

```
string greeting = "Hello, C#";
```

We have just declared the variable **greeting** of type **string** whose content is the text phrase "**Hello, C#**". The representation of the content in the string looks closely to this:

The internal representation of the class is quite simple – an array of characters. We can avoid the usage of the class by declaring a variable of type **char[]** and fill in the array's elements character by character. However, there are some disadvantages too:

1. Filling in the array happens character by character, not at once.
2. We should know the length of the text in order to be aware whether it will fit into the already allocated space for the array.
3. The text processing is manual.

The String Class: Universal Solution?

The usage of **System.String** is not the ideal and universal solution – sometimes it is appropriate to use different character structures.

In C# we there are other classes for text processing – we will become familiar with some of them later in this chapter.

The type **string** is more special from other data types. It is a class and as such it complies with the principles of object-oriented programming. Its values are stored in the dynamic memory (**managed heap**), and the variables of type **string** keeps a **reference** to an object in the heap.

Strings are Immutable

The **string** class has an important feature – the character sequences stored in a variable of the class are never changing (**immutable**). After being assigned once, the content of the variable does not change directly – if we try

to change the value, it will be saved to a new location in the dynamic memory and the variable will point to it. Since this is an important feature, it will be illustrated later.

Strings and Char Arrays

Strings are very similar to the char arrays (**char[]**), but unlike them, they **cannot be modified**. Like the arrays, they have properties such as **Length**, which returns the length of the string and allows access by index. Indexing, as it is used in arrays, takes indices from **0** to **Length-1**. Access to the character of a certain position in a string is done with the operator **[]** (indexer), but it is allowed only to read characters (and not to write to them):

```
string str = "abcde";
char ch = str[1]; // ch == 'b'
str[1] = 'a'; // Compilation error!
ch = str[50]; // IndexOutOfRangeException
```

Strings – Simple Example

Let's give an example for using variables from the type **string**:

```
string message = "This is a sample string message.";

Console.WriteLine("message = {0}", message);
Console.WriteLine("message.Length = {0}", message.Length);

for (int i = 0; i < message.Length; i++)
{
  Console.WriteLine("message[{0}] = {1}", i, message[i]);
}
// Console output:
// message = This is a sample string message.
// message.Length = 31
// message[0] = T
// message[1] = h
// message[2] = i
// message[3] = s
// …
```

Please note the string value – the quotes are not part of the text, they are enclosing its value. The example demonstrates how to print a string, how to extract its length and how to extract the character from which it is composed.

Strings Escaping

As we already know, if we want to use quotes into the string content, we must put a slash before them to identify that we consider the quotes character itself and not using the quotation marks for ending the string:

```
string quote = "Book's title is \"Intro to C#\"";
// Book's title is "Intro to C#"
```

The quotes in the example are part of the text. They are added in the variable by placing them after the escaping character backslash (****). In this way the compiler recognizes that the quotes are not used to start or end a string, but are a part of the data. Displaying special characters in the source code is called **escaping**.

Declaring a String

We can declare variables from the type string by the following rule:

```
string str;
```

Declaring a string represents a variable declaration of type **string**. This is not equivalent to setting a variable and allocating memory for it! With the declaration we inform the compiler that the variable **str** will be used and the expected type for it is **string**. We do not create a variable in the memory and it is not available for processing yet (value is **null**, which means no value).

Creating and Initializing a String

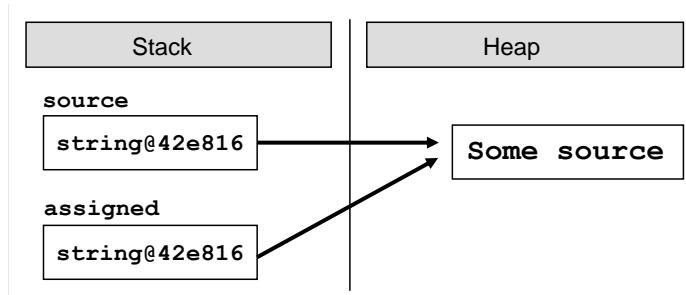
In order to process the declared string variable, we must create it and initialize it. Creating a variable of certain class (also known as **instantiating**) is a process associated with the allocation of the dynamic memory area (the heap). Before setting a specific value to the string, its value is **null**. This can be confusing to the beginner programmers: uninitialized variables of type **string** do not contain empty values, it contains the special value **null** – and each attempt for manipulating such a string will generate an error (exception for access to a missing value **NullReferenceException**)!

We can initialize variables in the following three ways:

1. By assigning a string literal.
2. By assigning the value of another string.
3. By passing the value of an operation which returns a string.

Setting a String Literal

Setting a string literal means **to assign a predefined textual content** to a variable of type **string**. We use this type of initialization, when we know the value that must be stored in the variable. Example for setting a string literal:

```
string website = "http://www.wikipedia.org";
```

In this example we created the variable **website** with value the above stated string literal.

Assigning Value of Another String

Assigning a value is equivalent to directing a **string** value or a variable to a variable of type **string**. An example is the following code snippet:

```
string source = "Some source";
string assigned = source;
```

First, we declare and initialize the variable **source**. Then the variable **assigned** takes the value of **source**. Since the **string** class is a **reference type**, the text "**Some source**" is stored in the dynamic memory (heap) on an address defined by the first variable.

In the second line we redirect the variable **assigned** to the same place, which the other variable points to. In this way the two objects receive the same address in dynamic memory and hence the same value.

The change of either variable will affect **only itself** because of the immutability of the type **string**, as when a change occurs, a copy of the changed string will be created. This is not true for the rest of the reference types (the normal, mutable types) because with them the changes are made directly in the address in memory and all references point to this changed address.

Passing a String Expression

The third option to initialize a string is to **pass the value of a string expression** or operation, which returns a string result. This can be a result from a method, which validates data; adding together the values of a number of constants and variables; transforming an existing variable, etc.

Example of an expression, which returns a string:

```
string email = "some@gmail.com";
```

```
string info = "My mail is: " + email;
// My mail is: some@gmail.com
```

The **info** variable has been created from the **concatenation** of literals and a variable.

Reading and Printing to the Console

Let's now take a look at the ways of reading strings, entered by the user and how we print strings to the console.

Reading Strings

Reading strings can be accomplished through the methods of the well-known **System.Console** class:

```
string name = Console.ReadLine();
```

In this example we read from the console the input data through the method **ReadLine()**. It waits for the user to input a value and to press [Enter]. After pressing the [Enter] key the variable **name** will contain the input name typed at the console (read from the keyboard).

What can we do after the variable has been created and it has a value itself? We can use it, for example, in expressions with other strings, to pass it as a method's parameter, to write it in text documents, etc. First, we can write it to the console in order to be sure that the data has been correctly read.

Printing Strings

Taking the data to the standard output is made also by the well-known class **System.Console**:

```
Console.WriteLine("Your name is: " + name);
```

By using the method **WriteLine(…)** we are getting the message "**Your name is:** " followed by the value of the **name** variable. After the end of the message a new line character is added. If we want to run away from the new line, which means the messages will appear at one and the same line then we use the method, **Write(…)**.

We can refresh our knowledge on the **System.Console** class from the chapter "Console Input and Output".

Strings Operations

After getting familiar with the strings semantics and how we can create and print them, next comes to learn how to deal with them and how to process

them. The C# language gives us a number of operations ready for use, which we will use for manipulating the strings.

Comparing Strings in Alphabetical Order

There are **many ways to compare strings** and depending on what exactly we need in the particular case, we can take advantage of the various features of the **string** class.

Comparison for Equality

If the requirements are to **compare the two strings** in order to determine whether their values are **equal or not**, the most convenient method is the **Equals(…)**, which works equivalently to the **operator ==**. It returns a Boolean result with either **true** value, if the strings have the same values, or **false** value, if they are different. The method **Equals(…)** checks letter by letter for equality of string values, as it makes distinction between small and capital letters, i.e. comparing the "c#" and "C#" with the **Equals(…)** method will return the value **false**. Consider the following example:

```
string word1 = "C#";
string word2 = "c#";
Console.WriteLine(word1.Equals("C#"));
Console.WriteLine(word1.Equals(word2));
Console.WriteLine(word1 == "C#");
Console.WriteLine(word1 == word2);

// Console output:
// True
// False
// True
// False
```

In practice, we often are interested of only the actual text content when comparing two strings, regardless of the **character casing** (uppercase / lowercase). To ignore the difference between small and capital letters in string comparison we can use the method **Equals(…)** with the parameter **StringComparison.CurrentCultureIgnoreCase**. So now in the same example of comparing "C#" with "c#" the method will return the value **true**:

```
Console.WriteLine(word1.Equals(word2,
  StringComparison.CurrentCultureIgnoreCase));
// True
```

StringComparison.CurrentCultureIgnoreCase is a constant of the enumerated type **StringComparison**. What is enumerated type and how we can use it, we will learn in the chapter "Defining Classes".

Comparing Strings in Alphabetical Order

It has become clear how we compare strings for equality, but how we are going to establish the **lexicographical order** of several strings? If we try to use the operators **<** and **>** which work great for comparing numbers, we find out that they **cannot be used for strings**.

If you want to compare two words and get information which one of them is before the other according to their **alphabetical order** of letters, here comes the method **CompareTo(…)**. It allows us to compare the values of two strings in order to determine their lexicographical order. In order two strings to have the same values, they must have the same length (number of characters) and the all their characters should match accordingly. For example, the strings "give" and "given" are different because they differ in their lengths, and "near" and "fear" differ in their first character.

The method **CompareTo(…)** from the **String** class returns a negative value, 0 or positive value depending on the lexical order of the two compared strings. A negative value means that the first string is **lexicographically** before the second, zero means that the two strings are equal and positive value means that the second string is lexicographically before the first. To clarify better how to compare strings lexicographically, let's go through a few examples:

```
string score = "sCore";
string scary = "scary";

Console.WriteLine(score.CompareTo(scary));
Console.WriteLine(scary.CompareTo(score));
Console.WriteLine(scary.CompareTo(scary));

// Console output:
// 1
// -1
// 0
```

The first experiment is called the method **CompareTo(…)** of the string **score**, as passed parameter is the variable **scary**. The first digit returns equal sign. Because the method **does not ignore** the casing of small and capital letters, it finds mismatch in the second character (in the first string it is "C", while in the second it is "c"). This is enough to determine the arrangement of strings and **CompareTo(…)** returns +1. Calling the same method with swapped places of the strings returns -1, because then the starting point is the string **scary**. His final call returns a logical 0, because we compare **scary** with itself.

If we have to **compare the strings lexicographically**, namely to **ignore the letters casing**, then we could use **string.Compare(string strA, string strB, bool ignoreCase)**. This is a static method, which works in the same way as **CompareTo(…)**, but it has an **ignoreCase** option for ignoring the casing of capital and small letters. Let's look at the method in action:

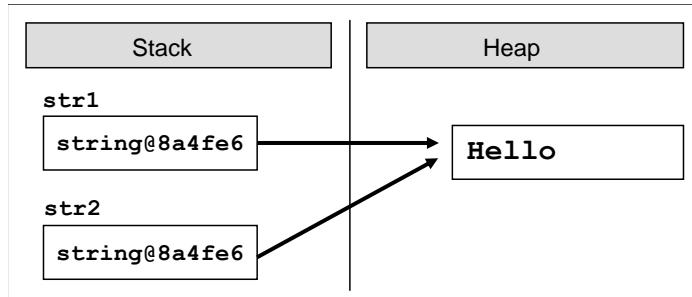
```
string alpha = "alpha";
string score1 = "sCorE";
string score2 = "score";

Console.WriteLine(string.Compare(alpha, score1, false));
Console.WriteLine(string.Compare(score1, score2, false));
Console.WriteLine(string.Compare(score1, score2, true));
Console.WriteLine(string.Compare(score1, score2,
  StringComparison.CurrentCultureIgnoreCase));
// Console output:
// -1
// 1
// 0
// 0
```

In the last example the method **Compare(…)** takes as a third parameter **StringComparison.CurrentCultureIgnoreCase** – already well-known from the method **Equals(…)** through which we can also compare strings, without having to register the difference between the small and capital letters.

Please note that according to the methods **Compare(…)** and **CompareTo(…)** the **small letters are lexicographically before the capital ones**. The correctness of this rule is quite controversial as in the Unicode table the capital letters are before the small ones. For example due to the standard Unicode, the letter "**A**" has a code 65, which is smaller than the code of the letter "**a**" (97).

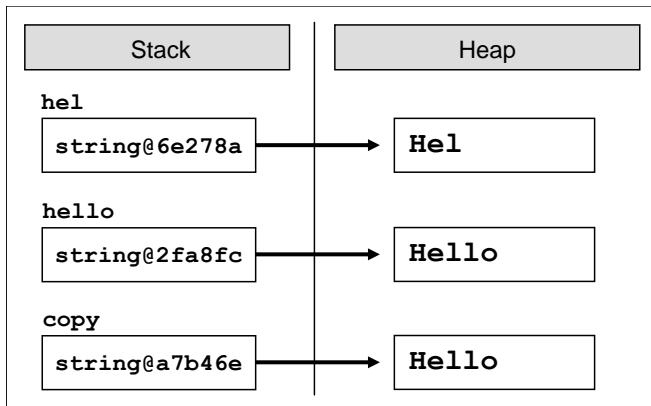
When you want just to consider whether the values of two strings are equal or not, please use the method Equals(…) or the operator ==. The methods CompareTo(…) and string.Compare(…) are designed to be used when the lexicographical order is needed.

Therefore, you should consider that the **lexicographical comparison does not follow the letter arrangement in the Unicode table**. Other abnormalities can also be caused by special features of the current culture. For some languages like German the characters "**ss**" and "**ß**" are considered equal. For example the words "Straße" and "Strasse" are considered the same by **CompareTo(…)** and equal when compared through the **==** operator:

```
string first = "Straße";
string second = "Strasse";

Console.WriteLine(first == second); // False
Console.WriteLine(first.CompareTo(second)); // 0 – equal strings
```

The == and != Operators

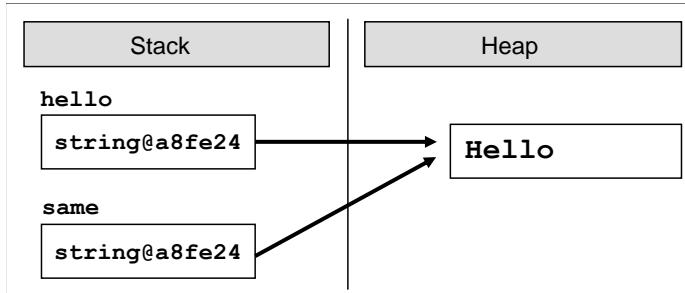
In the C# language the operators **==** and **=!** work for strings through an internal calling of **Equals(…)**. We will go through some examples for using those two operators with variables from the string type:

```
string str1 = "Hello";
string str2 = str1;

Console.WriteLine(str1 == str2);
// Console output:
// True
```

The comparison of matching strings **str1** and **str2** returns **true**. This is a fully expected result, since the target variable **str2** is pointed to the dynamic memory that is reserved for the variable **str1**. Thus, both variables have the same address and the check for equality returns true. Presented is how the memory looks like with the two variables:

Let’s look at another example:

```
string hel = "Hel";
string hello = "Hello";
string copy = hel + "lo";

Console.WriteLine(copy == hello);
// True
```

Pay attention to the **comparison** between the strings **hello** and **copy.** The first variable takes directly the value "**Hello**". The second takes its value as a result of joining a variable with literal, and the final result is equivalent to the value of the first variable. At this stage the two variables point to different areas of memory, but the contents of the memory blocks are identical. The comparison made with the operator **==** returns a result **true**, although both variables point to different areas of memory.

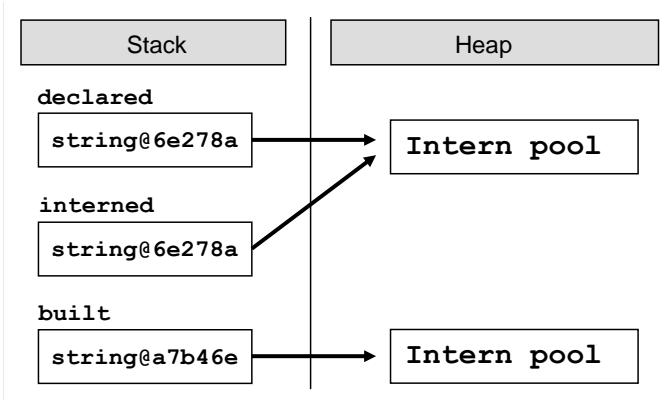
Here is how the memory looks like at this point:

Memory Optimization for Strings (Interning)

Let's consider the following example:

```
string hello = "Hello";
string same = "Hello";
```

Let's create a variable with value "**Hello**". We also create a second variable assigning it a value the same literal. It is logical when creating the variable **hello**, to allocate space in the heap, to write its value and the variable to point to that location. When creating the **same** a new place to record should be allocated too, the value should be written and the reference to the memory should be directed.

But the truth is that there is an optimization in the C# compiler and in CLR, which **saves the memory from creating duplicated strings**. This optimization is called **strings interning** and thanks to it the two variables in the memory will be pointed to the same common block of memory. This reduces the memory space usage and optimizes certain operations such as comparing two completely matching strings. They are written in the memory in the following way:

When we initialize a variable of type **string** with a string literal, the memory checks invisibly for us whether this value already exists. If the value already exists, the new variable is simply pointed to it. If not, a new block of memory is allocated, the value is stored in it and the reference is changed to point to

the new block. The **string interning** in .NET is possible because strings are **immutable by design** and it is not likely that the memory block referenced by several string variables will simultaneously be changed by someone.

When not initializing the strings with literals, no interning is used. However, if we want to use interning specifically, we can make it through the use of the method **Intern(…)**:

```
string declared = "Intern pool";
string built = new StringBuilder("Intern pool").ToString();
string interned = string.Intern(built);

Console.WriteLine(object.ReferenceEquals(declared, built));
Console.WriteLine(object.ReferenceEquals(declared, interned));
// Console output:
// False
// True
```

Here is the memory situation at this moment:

In the example we used the static method **Object.ReferenceEquals(…)**, which compares two objects in memory and returns whether they point to the same memory block. We used the class **StringBuilder**, which serves to efficiently build strings. When and how to use **StringBuilder** we will explain in details shortly, but now let's get familiar with the basic operations on strings.

Operations for Manipulating Strings

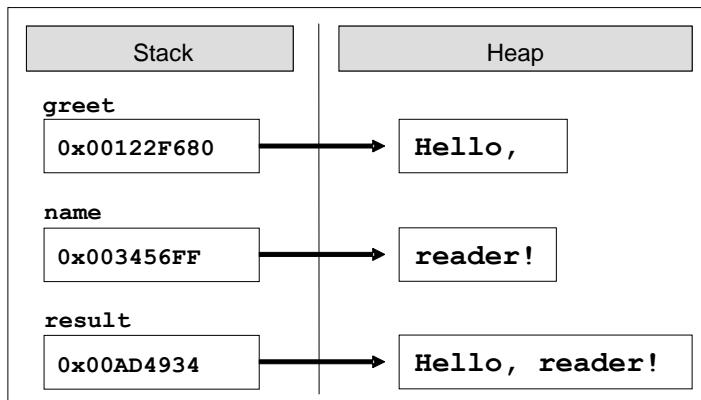
Once we got familiar with the fundamentals of strings and their structure, the next thing to explore are the tools for their processing. We will review string **concatenation**, **searching** in a string, extracting **substrings**, change the character **casing**, **splitting** a string by separator and other string operations that will help us solve various problems from the everyday practice.

Strings are immutable! Any change of a variable of type string creates a new string in which the result is stored. Therefore, operations that apply to strings return as a result a reference to the result.

It is possible to process strings without creating new objects in the memory every time a modification is made but for this purpose the class **StringBuilder** should be used. We will introduce it a bit later.

Strings Concatenation

Gluing two strings and obtaining a new one as a result is called **concatenation**. It could be done in several ways: through the method **Concat(…)** or with the operators **+** and **+=**.

Example of using the method **Concat(…)**:

```
string greet = "Hello, ";
string name = "reader!";
string result = string.Concat(greet, name);
```

By calling the method, we will concatenate the string variable **name**, which is passed as an argument, to the string variable **greet**. The result string will be the text "**Hello, reader!**".

The second way for concatenation is via the operators **+** and **+=**. Then the above example can be implemented in the following way:

```
string greet = "Hello, ";
string name = "reader!";
string result = greet + name;
```

In both cases those variables will be presented in the memory as follows:

Please note that string concatenation **does not change the existing strings** but returns a new string as a result. If we try to concatenate two strings without storing them in a variable, the changes would not be saved. Here is a typical mistake:

```
string greet = "Hello, ";
string name = "reader!";
string.Concat(greet, name);
```

In the given example the two variables are concatenated but the result of it has not been saved anywhere, so it is lost:

If we want to add a value to an existing variable, for example the variable **result**, we can do it with the well-known code:

```
result = result + " How are you?";
```

In order to avoid the double writing of the above declared variable, we can use the operator **+=**:

```
result += " How are you?";
```

The result will be the same in both cases: "**Hello, reader! How are you?**".

We can **concatenate other data with strings**. Any data, which can be presented in a text form, can be appended to a string. Concatenation is possible with numbers, characters, dates, etc. Here is an example:

```
string message = "The number of the beast is: ";
int beastNum = 666;
string result = message + beastNum;
// The number of the beast is: 666
```

As we understood from the above example, there is no problem in concatenating strings with other data, which is not from a **string** type. Let's have another full example for string concatenation:

```
public class DisplayUserInfo
{
  static void Main()
  {
    string firstName = "John";
    string lastName = "Smith";
    string fullName = firstName + " " + lastName;

    int age = 28;
    string nameAndAge = "Name: " + fullName + "\nAge: " + age;
```

```
        Console.WriteLine(nameAndAge);
    }
}
// Console output:
// Name: John Smith
// Age: 28
```

Switching to Uppercase and Lowercase Letters

Sometimes we need to **change the casing of a string** so that all the characters in it to be entirely **uppercase or lowercase**. The two methods that would work best in this case are **ToLower(…)** and **ToUpper(…)**. The first converts all capital letters to small ones:

```
string text = "All Kind OF LeTTeRs";

Console.WriteLine(text.ToLower());
// all kind of letters
```

The example shows that all capital letters of the text change their casing and the entire text goes in **lowercase**. Such a shift to lowercase is convenient for storing usernames in various online systems. Upon registration the users may use a mixture of uppercase and lowercase letters, but the system can then make them all small to unify them and to avoid matches on points with differences in the casing.

Here is another example. We want to compare entered by the user input but we are not sure exactly how it was written – in small or capital letters or mixed. One possible approach is to standardize capitalization and compare it with the constant defined by us. Thus, we **make no distinction of small and capital letters**. For example, if we have a user input panel where we enter name and password and it does not matter if the password is written with capital letters or small, we can make a similar check on the password:

```
string pass1 = "PasswoRd";
string pass2 = "PaSSwoRD";
string pass3 = "password";

Console.WriteLine(pass1.ToUpper() == "PASSWORD");
Console.WriteLine(pass2.ToUpper() == "PASSWORD");
Console.WriteLine(pass3.ToUpper() == "PASSWORD");

// Console output:
// True
```

```
// True
// True
```

In the example we are comparing three passwords with the same content but with a different casing. When checking their contents, always verify if it equals to the string "**PASSWORD**" (letter by letter). Of course, we could do the above verification and by the method **Equals(…)** in the version with ignoring the character casing, which we already discussed.

Searching for a String within Another String

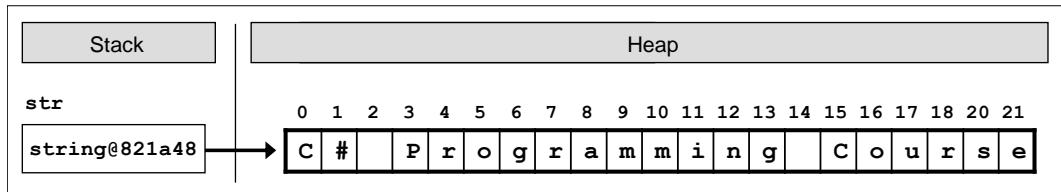
When we have a string with a specified content, it is often necessary to process only a part of its value. The .NET platform provides us with two methods to **search a string within another string**: **IndexOf(…)** and **LastIndexOf(…)**. They search into the string and check whether the passed as a parameter substring occurs in its content. The result of those methods is an integer. If the result is not a negative value, then this is the position where the first character of the substring is found. If the method returns value of -1, it means that the substring was not found. Remember that in C# indexing into strings start from 0.

The methods **IndexOf(…)** and **LastIndexOf(…)** search the contents of the text sequence, but in a different direction. The search with the first method starts from the beginning of the string towards the end, while the second method – the search is done backwards. If we are interested in the first encountered match, then we use **IndexOf(…)**. If we want to search the string from its end (for example to detect the last dot in a file name or the last slash in an URL address), then we use **LastIndexOf(…)**.

When calling **IndexOf(…)** and **LastIndexOf(…)** a second parameter could be passed, which will specify the position, which the searching should start from. This is useful if we want to search part of a string, not the entire string.

Searching into a String – Example

Let's consider an example with the **IndexOf(…)** method:

```
string book = "Introduction to C# book";
int index = book.IndexOf("C#");

Console.WriteLine(index);
// index = 16
```

In the example, the variable **book** has a value "**Introduction to C# book**". The search for the substring "**C**" in this variable will return the value 16, because the substring will be found and the first character "**C**" of the searched word is in 16th position.

Searching with IndexOf(…) – Example

Let's look into great details one more example for searching for a separate characters or strings in a text:

```
string str = "C# Programming Course";

int index = str.IndexOf("C#"); // index = 0
index = str.IndexOf("Course"); // index = 15
index = str.IndexOf("COURSE"); // index = -1
index = str.IndexOf("ram");    // index = 7
index = str.IndexOf("r");      // index = 4
index = str.IndexOf("r", 5);   // index = 7
index = str.IndexOf("r", 10);  // index = 18
```

Look how the string we are searching looks like in the memory:

If we look at the results of the third search, we will note that the search for the word "**COURSE**" in the text returned a result of -1, i.e. no match has been found. Although the word is in the text, it has been written in a different case of letters. The methods **IndexOf(…)** and **LastIndexOf(…)** distinguish between uppercase and lowercase letters. If we want to ignore this difference, we can write text in a new variable and turn it to a text with entirely lower or entirely uppercase, and then we can perform the search in it, independently from the letters casing.

Finding All Occurrences of a Substring – Example

Sometimes we want to **find all occurrences of a particular substring within another string**. Using both methods with only one searched string passed as an argument would not work for us, because it will always return only the first occurrence of the substring. We can pass a second parameter for an index that indicates the starting position from which the searching should begin. Of course, we need to loop through it in order to move from the first occurrence of the searched string to the next, to the next, and the next, etc., until the last one.

Here is an example how we can use the method **IndexOf(…)** by a given word and start index: finding all occurrences of the word "C#" in a given text:

```
string quote = "The main intent of the \"Intro C#\"" +
  " book is to introduce the C# programming to newbies.";
```

```
string keyword = "C#";
int index = quote.IndexOf(keyword);

while (index != -1)
{
  Console.WriteLine("{0} found at index: {1}", keyword, index);
  index = quote.IndexOf(keyword, index + 1);
}
```

The first step is to make a search for the keyword "C#". If the word is found in the text (i.e. the returned value is different than -1), it prints it on the console and we continue our search rightwards, starting from the position on which we have found the word plus one. We repeat this operation until **IndexOf(…)** returns value -1.

Note: If we miss setting an initial index, then the search will always start from the beginning and will return one and the same value. This will lead to **hanging of the program**. If we search directly from the index without adding plus one each time, we will come across again and again to the last result, whose index we have already found. Therefore, proper search of the next result should start from a starting position **index + 1**.

Extracting a Portion of a String

For now we know how to check whether a substring occurs in a text and which are the occurrence positions. But how can we **extract a portion of a string** in a separate variable?

The solution of this problem is the method **Substring(…)**. By using it, we can extract a **part of the string (substring)** by a given starting position in the text and its length. If the length is omitted, a portion from the text will be extracted, starting from the initial position to the string's end.

Presented is an example of **extracting a substring from a string**:

```
string path = "C:\\Pics\\CoolPic.jpg";
string fileName = path.Substring(8, 7);
// fileName = "CoolPic"
```

We manipulate the variable **path**. It contains the path to a file from our file system. To assign the file name to a new variable, we use **Substring(8, 7)** and take a sequence of 7 characters starting from the 8th position, i.e. character positions from 8 to 14 inclusively.

Calling the method Substring(startIndex, length), extracts a substring from a string, which is located between startIndex and (startIndex + length – 1) inclusively. The character at

	the position `startIndex + length` is not taken into consideration! For example, if we point `Substring(8, 3)`, the characters between index 8 and 10 inclusively will be extracted.

Here are presented the characters, which form the text from which we extract a substring:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	**:**	****	**P**	**i**	**c**	**s**	****	**C**	**o**	**o**	**l**	**P**	**i**	**c**	**.**	**j**	**p**	**g**

Sticking to the scheme, the method that has been called must write the characters from the positions 8 to 14 (as the last index is not included), namely "**CoolPic**".

Extracting a File Name and File Extension – Example

Let's consider a more interesting task. How can we print the **filename and its extension** from given full path to a file in Windows-based file system? As we know how the path is recorded in the file system, we can proceed with the following plan:

- Looking for the **last backslash** in the text;
- Keeping the **position** of the last backslash;
- Extracting the substring starting from **the obtained position +1**;

Let's consider again the example of the well-known **file path**. If we have no information about the exact contents of the variable, but we know that it contains a file path, we can stick to the above scheme:

```
string path = "C:\\Pics\\CoolPic.jpg";
int index = path.LastIndexOf("\\");
// index = 7
string fullName = path.Substring(index + 1);
// fullName = "CoolPic.jpg"
```

Splitting the String by a Separator

One of the most flexible methods for working with strings is **`Split(…)`**. It allows us to **split a string by a separator** or an array of possible separators. For example, we can process a variable, which has the following content:

```
string listOfBeers = "Amstel, Heineken, Tuborg, Becks";
```

How can we split each beer in a separate variable or extract all beers in an array? At first glance it may seem difficult – we must seek with **`IndexOf(…)`** for a special character, then to extract a substring with **`Substring(…)`**, to

iterate all this in a loop and to write the result in a variable. Since the splitting of a string by a separator is a main task of text processing, ready to use methods for it can be found in .NET Framework.

Splitting Strings by Multiple Separators – Example

The easiest and more flexible method for resolving this issue is the following:

```
char[] separators = new char[] {' ', ',', '.'};
string[] beersArr = listOfBeers.Split(separators);
```

Using the built-in functionality of the method **Split(…)** from the class **String**, we will split the contents of a given string by array of characters – separators, which are passed as an argument of the method. All substrings among which are space, comma or dot will be removed and stored in the **beersArr** array.

If we iterate the array and print its elements one by one, the result will be: "Amstel", "", "Heineken", "", "Tuborg", "" and "Becks". We get 7 results, instead of the expected 4. The reason is that during the text splitting, three substrings are found which contain two separator characters one next to the other (for example a comma, followed by a space). In this case the empty string between the two separators is also part of the returned result.

How to Remove the Empty Elements after Splitting?

If we want to ignore the empty strings from the splitting results, one possible solution is to make checks on their printing:

```
foreach (string beer in beersArr)
{
   if (beer != "")
   {
      Console.WriteLine(beer);
   }
}
```

But this approach does not remove the empty strings from the array. It just does not print them. So we can change the arguments we are passing to the method **Split(…)**, by passing a special option:

```
string[] beersArr = listOfBeers.Split(
   separators, StringSplitOptions.RemoveEmptyEntries);
```

After this change, the **beersArr** array will contain 4 elements – the 4 words from the **listOfBeers** variable.

When splitting strings and adding as a second parameter the constant StringSplitOptions.RemoveEmptyEntries we instruct

	the method Split(…) to work in the following way: "Return all substrings from the variable that are split by given list of separators. If you meet two or more neighboring separators, consider them as one."

Replacing a Substring

The text processing in .NET Framework provides ready methods for **replacing a substring with another**. For example, if we have made one and the same technical mistake when typing the email address of a user in an official document, we can replace it by using the method **Replace(…)**:

```
string doc = "Hello, some@gmail.com, " +
  "you have been using some@gmail.com in your registration.";
string fixedDoc =
  doc.Replace("some@gmail.com", "john@smith.com");
Console.WriteLine(fixedDoc);

// Console output:
// Hello, john@smith.com, you have been using
// john@smith.com in your registration.
```

As it can be seen from the example, the method **Replace(…)** **replaces all occurrences** of a given substring with another substring, not just the first.

Regular Expressions

The regular expressions are a powerful tool for text processing and allow searching matches by a **pattern**. An example for a pattern is **[A-Z0-9]+**, which means not an empty series of capital Latin letters and numbers.

Regular expressions make text processing **easier and more accurate**: extracting some resources from texts, searching for phone numbers, finding email addresses in a text, splitting all the words in a sentence, data validation, etc.

Regular Expressions – Example

If we have an official document that is used only in the office and it contains a lot of personal data, then we should censor it before sending it to the client. For example, we can censor all mobile numbers and replace them with asterisks. By using **regular expressions**, this could be done as follows:

```
string doc = "Smith's number: 0898880022\nFranky can be " +
  "found at 0888445566.\nSteven's mobile number: 0887654321";
string replacedDoc = Regex.Replace(
  doc, "(08)[0-9]{8}", "$1********");
```

```
Console.WriteLine(replacedDoc);
// Console output:
// Smith's number: 08********
// Franky can be found at 08********.
// Steven' mobile number: 08********
```

Explaining the Arguments of Regex.Replace(…)

In the above code fragment by using a regular expression, we find all the phone numbers specified in the text and replace them by a pattern. We use the class **System.Text.RegularExpressions.Regex**, which is intended for use with regular expressions in .NET Framework. The variable, which imitates the document text, is **doc**. Several names of customers are recorded there. If we want to protect the contacts from an improper use and wish to censor the phone numbers, then we can replace all mobile phones with asterisks. Assuming that the phones are saved in the following format: "**08 + 8 digits**", the method **Regex.Replace(…)** finds all matches by a given format and replaces them with: "**08**********".

The regular expression that finds all of the numbers is the following: "**(08)[0-9]{8}**". It finds all substrings in the text, constructed by the constant "**08**" and followed exactly by 8 characters ranging from 0 to 9. The example can be further improved by selecting the numbers only from a given mobile operator, for phones on foreign networks, etc., but in this case we used the simplified version.

The literal "**08**" is surrounded by parentheses. They serve for forming a separate group in the regular expression. The groups can be used for handling only a certain part of the expression instead of the entire expression. In our example, the group is used in the substitution. Through it, the founded matches are replaced by the pattern "**$1**********", i.e. the text which was found in the first group of the regular expression (**$1**) + 8 consecutive asterisks for censorship. As the defined group is always a constant (08), so the text replaced will always be: **08 ********.**

This chapter is not intended to explain in details **how to use regular expressions in .NET Framework**, as it is a huge and complex field, but only to turn the reader's attention that the regular expressions exist and they are a powerful tool for text processing. Anyone who wants to learn more, can search for articles, books and tutorials in order to learn how to construct regular expressions, how to look for matches, how validation is made, how to make substitutions by patterns, etc. In particular, we recommend you to visit the websites http://www.regular-expressions.info and http://regexlib.com. More information about the classes in .NET Framework for working with regular expressions can be found at: http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex%28VS.100%29.aspx.

Removing Unnecessary Characters at the Beginning and at the End of a String

When entering text in a file or to the console, you can find sometimes some "parasitic" spaces (**white-space**) at the beginning or at the end of the text – some other space or a tab that cannot be observed at first glance. This may not be essential but if we do not validate the user data, there would be a problem in terms of checking the contents of the input information. In order to solve this problem we can use the method **Trim()**. It is responsible for eliminating (**trimming**) the white spaces at the beginning or at the end of a string. The white spaces can be spaces, tabs, line breaks etc.

Let's assume in the variable **fileData** we have read the contents of a file where is written a name of a student. There may have emerged parasitic spaces when writing the text or reversing it from one format to another. In that case the variable will look the following way:

```
string fileData = "   \n\n      David Allen      ";
```

If we print the contents to the console, we get two blank lines followed by some spaces, the requested name and some additional spaces at the end. We can reduce the information just to the required name, in the following way:

```
string reduced = fileData.Trim();
```

When we print the information to the console for the second time, the content will be "**David Allen**", without any unwanted white spaces.

Removing Unnecessary Characters by a Given List

The method **Trim(…)** can accept an array of characters, which we want to remove from the string. We can make it in the following way:

```
string fileData = "   111 $  %    David Allen  ### s   ";
char[] trimChars = new char[] {' ', '1', '$', '%', '#', 's'};
string reduced = fileData.Trim(trimChars);
// reduced = "David Allen"
```

Again, we get the desired result "**David Allen**".

Please note that we must list all the characters we want to eliminate, including the empty spaces (spaces, tabs, new line, etc.). Without a `' '` in the array `trimChars`, we would not get the desired result!

If we want to remove the white spaces only at the beginning or in end of the string, we can use the methods **TrimStart(…)** and **TrimEnd(…)**:

```
string reduced = fileData.TrimEnd(trimChars);
// reduced = "   111 $  %    David Allen"
```

Constructing Strings: the StringBuilder Class

As explained above, strings in C# are immutable. This means that any adjustments applied to an existing string do not change it but return a new string. For example, using methods like **Replace(…)**, **ToUpper(…)**, **Trim(…)** do not change the string, which they are called for. They allocate **a new area in the memory** where the new content is saved. This behavior has many advantages but in some cases can cause performance problems.

Strings Concatenation in a Loop: Never Do This!

Serious **performance problems** may be encountered when trying to concatenate strings in a loop. The problem is directly related to the strings handling and dynamic memory, which is used to store them. To understand why we have **poor performance when concatenating strings in a loop**, we must first consider what happens when using operator "**+**" for strings.

How Does the String Concatenation Works?

We already got familiar with the ways to do string concatenation in C#. Let's now examine **what happens in memory when concatenating strings**. Consider two variables **str1** and **str2** of type **string**, which have values of "Super" and "Star". There are two areas in the heap (dynamic memory) in which the values are stored. The task of **str1** and **str2** is to keep a reference to the memory addresses where our data is stored. Let's create a variable **result** and give it a value of the other two strings by concatenation. A code fragment for creating and defining the three variables would look like this:

```
string str1 = "Super";
string str2 = "Star";
string result = str1 + str2;
```

What will happen with the memory? Creating the variable **result** will allocate a new area in dynamic memory, which will record the outcome of the **str1 + str2**, which is "**SuperStar**". Then the variable itself will keep the address of the allocated area. As a result we will have three areas in memory and three references to them. This is convenient, but allocating a new area, recording a value, creating a new variable and referencing it in the memory is time-consuming process that would be a problem when repeated many times, typically inside a loop.

Unlike other programming languages, in C# is not necessary to manually dispose the objects stored in memory. There is a special mechanism called a **garbage collector (memory cleaning system)**, which takes care of clearing the unused memory and resources. The garbage collector is

responsible for disposing of objects in dynamic memory when they are no longer used. Creation of many objects containing multiple references in dynamic memory is bad, because it fills memory and then the garbage collector is automatically enforced to start execution. It takes quite some time and **slows the overall performance of the process**. Furthermore, transferring characters from one place to another in memory (when string concatenation is executed) is slow, especially if the strings are long.

Why Concatenating Strings in a Loop is a Bad Practice?

Assume that we have a task to store the numbers from 1 to 20,000 consecutively to each other in a variable of type **string**. How can we solve the problem with our already existing knowledge? One of the easiest ways for implementation is to create a variable that stores the numbers and execute a loop from 1 to 20,000 in which each number is concatenated to the variable. Implemented in C#, the solution would look like this:

```
string collector = "Numbers: ";
for (int index = 1; index <= 20000; index++)
{
  collector += index;
}
```

Execution of the above code will loop 20,000 times and after each iteration will add the current index to the **collector** variable. **collector**'s value after implementation would be: "Numbers: 12345678910111213141516…" (the numbers from 17 to 20,000 are replaced with dots because we don't have the space to write something that long here).

Probably you have not noticed the **delay** in the fragment's execution. Indeed, using concatenation in the loop has **delayed significantly** the normal calculation process. On an average PC (as of January 2012) the loop iteration takes **1-2 seconds**. The user of our program would be very skeptical if he has to wait a few seconds for something so simple such as concatenating the numbers from 1 to 20,000. Moreover, in this case 20,000 is just an example endpoint. What will be the delay if instead of 20,000 the user needs to concatenate numbers to 200,000? Try it!

Concatenating in Loop of 200,000 Iterations – Example

Let's develop further the example above. First, we will change the endpoint of the loop from 20,000 to **200,000**. Second, in order to account properly the execution time, we will display on the console the current date and time before and after execution of the loop. Third, to see whether the variable contains the desired value, we will also display part of it on the console. If you want to make sure that the whole value is stored, you can remove the method **Substring(…)**, but the print itself in this case will take a long time.

The final version of the example would look like this:

```
class SlowNumbersConcatenator
{
   static void Main()
   {
      Console.WriteLine(DateTime.Now);

      string collector = "Numbers: ";
      for (int index = 1; index <= 200000; index++)
      {
         collector += index;
      }

      Console.WriteLine(collector.Substring(0, 1024));
      Console.WriteLine(DateTime.Now);
   }
}
```

When executing the example implementation on the console, the program starting date and time, the first 1024 characters of the variable and program completion date and time are displayed on the console. The reason to show only the first 1024 characters is that we want to measure only the calculation time without the time for printing the results. Printing the whole result will be time consuming. Let's see sample output from the execution:

```
C:\Windows\system32\cmd.exe
23.03.2013 23:25:34
Numbers: 12345678910111213141516171819202122232425262728293031
32333435363738394041424344454647484950515253545556575859606162636465666768697071727374757677787980818283848586878889909192939495969798991001011021031041051061071081091101111121131141151161171181191201211221231241251261271281291301311321331341351361371381391401411421431441451461471481491501511521531541551561571581591601611621631641651661671681691701711721731741751761771781791801811821831841851861871881891901911921931941951961971981992002012022032042052062072082092102112122132142152162172182192202212222232242252262272282292302312322332342352362372382392402412422432442452462472482492502512522532542552562572582592602612622632642652662672682692702712722732742752762772782792802812822832842852862872882892902912922932942952962972982993003013023033043053063073083093103113123133143153163173183193203213223233243253263273283293303313323333343353363373383393403413423433443453463473483493503513523533543553563573583593603613623633643653663673683693703713723733743
23.03.2013 23:31:09
Press any key to continue . . . _
```

Program start is marked with a green line and its end – with red. Note the execution time – about **5-6 minutes** (on our computer from January 2012)! Such a delay is unacceptable for such a task and will not only make the user nervous but will make him stop the program without waiting for it to end.

Processing Strings in the Memory

The problem with time-consuming Loop processing is related to the way strings work in memory. Each iteration creates a new object in the heap and point the reference to it. This process requires a certain physical time.

Several things happen at each step:

1. An area of memory is allocated for recording the next number concatenation result. This memory is used only temporarily while concatenating, and is called a **buffer**.

2. The old string is **moved** into the new buffer. If the string is long (say 500 KB, 5 MB or 50 MB), it can be **quite slow**!

3. Next number is **concatenated** to the buffer.

4. The buffer is **converted to a string**.

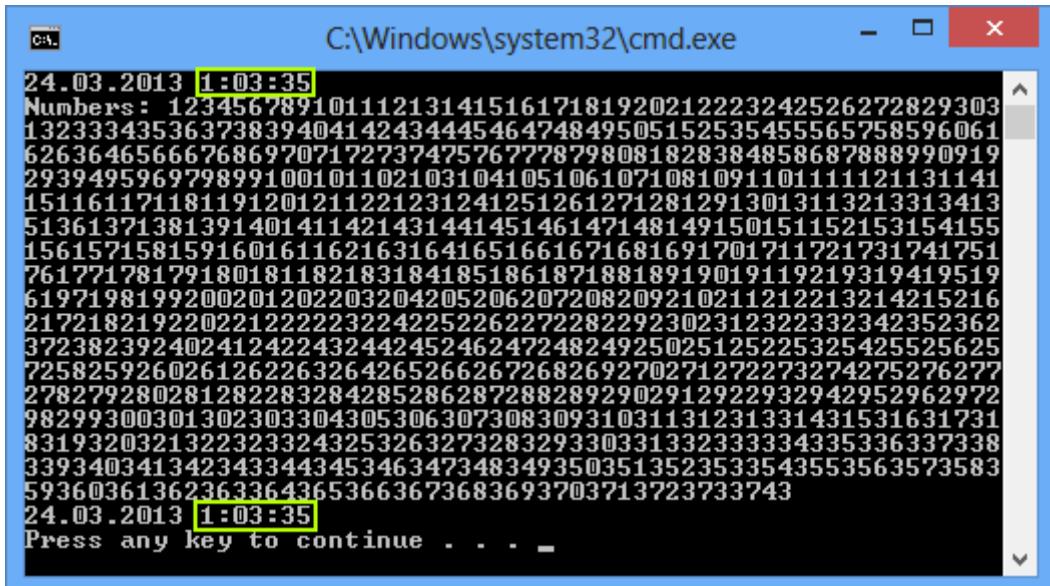
5. The old string and the temporary buffer become unused. Later they are **destroyed by the garbage collector**. This may also be a slow operation.

Much more elegant and appropriate way to concatenate strings in a Loop is using the **StringBuilder** class. Let's see how it works.

Building and Changing Strings with StringBuilder

StringBuilder is a class that serves to build and change strings. It **overcomes the performance problems** that arise when concatenating strings of type **string**. The class is built in the form of an array of characters and what we need to know about it is that the information in it can be freely changed. Changes that are required in the variables of type **StringBuilder**, are carried out in the same area of memory (buffer), which saves time and resources. Changing the content does not create a new object but simply changes the current.

Let's rewrite the code above in which we concatenated strings in a loop. If you remember, the operation previously took 5 minutes. Let's measure how long will take the same operation if we use **StringBuilder**:

```
class ElegantNumbersConcatenator
{
  static void Main()
  {
    Console.WriteLine(DateTime.Now);

    StringBuilder sb = new StringBuilder();
    sb.Append("Numbers: ");

    for (int index = 1; index <= 200000; index++)
```

```
        {
            sb.Append(index);
        }

        Console.WriteLine(sb.ToString().Substring(0, 1024));
        Console.WriteLine(DateTime.Now);
    }
}
```

This example is based on the previous one, with only minor adjustments. Return value is the same, but what about the execution time?

```
C:\Windows\system32\cmd.exe
24.03.2013 1:03:35
Numbers: 12345678910111213141516171819202122232425262728293031
132333435363738394041424344454647484950515253545556575859606162
626364656667686970717273747576777879808182838485868788899091
92939495969798991001011021031041051061071081091101111121131141
15116117118119120121122123124125126127128129130131132133134135
13613713813914014114214314414514614714814915015115215315415515
15615715815916016116216316416516616716816917017117217317417517
76177178179180181182183184185186187188189190191192193194195
61971981992002012022032042052062072082092102112122132142152162
21721821922022122222322422522622722822923023123223323423523623
37238239240241242243244245246247248249250251252253254255256257
72582592602612622632642652662672682692702712722732742752762772
27827928028128228328428528628728828929029129229329429529629729
98299300301302303304305306307308309310311312313314315316317318
83193203213223233243253263273283293303313323333343353363373383
3393403413423433443453463473483493503513523533543553563573583
59360361362363364365366367368369370371372373374
24.03.2013 1:03:35
Press any key to continue . . .
```

The time required to concatenate 200,000 characters with **StringBuilder** is now less than a second (perhaps few milliseconds)!

Reversing a String – Example

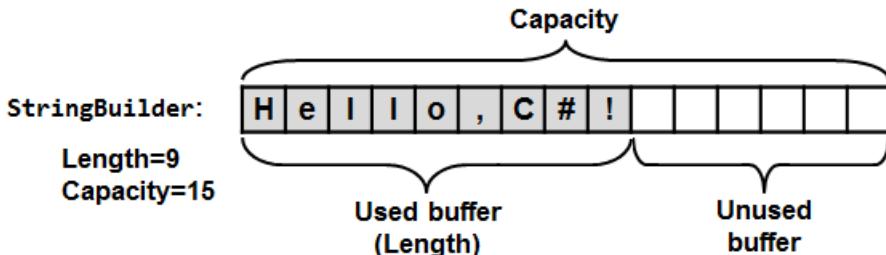
Consider another example: we want to reverse an existing string (backwards). For example, if we have the string "**abcd**", the returned result should be "**dcba**". We get the original string, iterate it backwards character by character and add each character to a variable of type **StringBuilder**:

```
public class WordReverser
{
    static void Main()
    {
        string text = "EM edit";
        string reversed = ReverseText(text);
```

```
        Console.WriteLine(reversed);

        // Console output:
        // tide ME
    }

    static string ReverseText(string text)
    {
        StringBuilder sb = new StringBuilder();
        for (int i = text.Length - 1; i >= 0; i--)
        {
            sb.Append(text[i]);
        }
        return sb.ToString();
    }
}
```

In this example we have a variable **text**, which contains the value "EM edit". We pass the variable to the **ReverseText(…)** method and set the new value in a variable named **reversed**. The method, in turn, iterates the characters of the variable in reverse order and stores them in a new variable of type **StringBuilder,** but now back ordered. Ultimately, the result is "tide ME".

How Does the StringBuilder Class Work?

The **StringBuilder** class is an implementation of a string in C#, but different than the class **String**. Unlike the already familiar for us strings, the objects of the **StringBuilder** class are not immutable, namely **edit operations do not require creating a new object** in the memory. This reduces the unnecessary transfer of data in memory when performing basic operations such as string concatenation.

StringBuilder keeps a buffer with a certain capacity (16 characters by default). The buffer is implemented as an array of characters that is provided to the developer by a user-friendly interface – methods that quickly and easily add and edit elements of the string. At any moment part of the characters in the buffer are used and the rest stay in reserve. This allows the addition to work very quickly. Other operations also operate faster than the class **string**, because the changes do not create a new object.

Once the internal buffer of the **StringBuilder** is full, it automatically is doubled (the internal buffer is resized to increase its capacity while its content is kept unchanged). **Resizing is a slow operation** but is happens rarely so the total performance is good. We will discuss this in more details in the chapter about "Algorithms Complexity".

Let's create an object of the **StringBuilder** class with 15 characters long buffer. We add the string: "**Hello, C#!**" to it and we get the following code:

```
StringBuilder sb = new StringBuilder(15);
sb.Append("Hello, C#!");
```

After creating the object and storing the value in it, the **StringBuilder** will look as follows:

Colored elements are the filled with our content part of the buffer. Normally, adding a new character to the variable does not create a new object in the memory but use the already allocated and unused space. If the entire capacity of the buffer is filled, then the buffer is doubled as we already explained.

StringBuilder – More Important Methods

The **StringBuilder** class provides us with a set of methods that help us to easily and efficiently edit text data and construct text. We met some of them in the examples. The most important are:

- **StringBuilder(int capacity)** – constructor with an initial capacity parameter. It may be used to set the buffer size in advance if we have estimates of the number of iterations and concatenations, which will be performed. This way we can save unnecessary dynamic memory allocations.
- **Capacity** – returns the buffer size (total number of used and unused positions in the buffer).
- **Length** – returns length of string saved in the variable (number of used positions in the buffer)
- Indexer **[int index]** – return the character stored in given position.
- **Append(…)** – appends string, number or other value after the last character in the buffer.
- **Clear(…)** – removes all characters from the buffer (deletes it).
- **Remove(int startIndex, int length)** – removes (deletes) string from the buffer with a given start position and length.

- **Insert(int offset, string str)** – inserts a string in a given start position (offset).
- **Replace(string oldValue, string newValue)** – replaces all occurrences of a given substring with another substring.
- **ToString()** – returns the **StringBuilder** object content as a **string** object.

Extracting All Capital Letters from a Text – Example

The next task is to **extract all capital letters from a text**. We can implement it in different ways – using an array, counter and filling the array with all capital letters found; creating an object of type **string** and concatenate capitals one by one to it; using the class **StringBuilder**.

Turning to the option of using an array, we have a problem: we do not know what will be array size, as we have no idea in advance how many are the capital letters in the text. We can create an array as large as the text, but thus wasting unnecessary space in memory and we must also maintain a counter that keeps where the array is full to.

Another option is to use a variable of type **string**. As we will iterate the whole text and concatenate all capital letters to the variable, probably we will lose efficiency again due to the strings concatenation.

StringBuilder: the Right Solution

The most viable solution to the task again is to use **StringBuilder**. We can start with an empty **StringBuilder**, iterate the letters of the given text character by character, verify that the current character is uppercase and concatenate the character at the end of our **StringBuilder**. Finally, we can return the final result by calling the **ToString()** method. Below is a sample implementation:

```
public static string ExtractCapitals(string str)
{
  StringBuilder result = new StringBuilder();
  for (int i = 0; i < str.Length; i++)
  {
    char ch = str[i];
    if (char.IsUpper(ch))
    {
      result.Append(ch);
    }
  }
  return result.ToString();
}
```

Calling **ExtractCapitals(…)** method and passing a specified text as a parameter to it, the return value is a string of all capital letters in the text, namely the initial string without all characters that are not capitalized. To check whether a character is uppercase we are using **char.IsUpper(…)** – a method from the standard .NET classes. You can view the **char** class documentation, because it offers other useful methods for handling characters.

String Formatting

.NET Framework provides the developer with mechanisms for formatting strings, numbers and dates. We have already met some of them in the chapter "Console Input and Output". Now we will extend our knowledge with methods for formatting and converting strings of the **string** class.

The ToString(…) Method

One of the interesting concepts in .NET is that practically every object of a class and primitive variables can be **presented as text**. This is done by the method **ToString(…)**, which is present in all .NET objects. It is implicit in the definition of the **object** class – the base class that all .NET data types inherit directly or indirectly. Thus the definition of the method appears in each class and we can use it to bring the content of each object in some text form.

The method **ToString(…)** is called automatically when we print objects from different classes to the console. For example, when printing dates the submitted date is converted to text by calling the **ToString(…)**:

```
DateTime currentDate = DateTime.Now;
Console.WriteLine(currentDate);
// Output: 01.02.2012 13:34:27 (depends on the culture settings)
```

When we pass **currentDate** as a parameter of the **WriteLine(…)** method, we don’t have an accurate statement that handles dates. The method has a particular implementation for all primitive types and strings. For all other objects **WriteLine(…)** calls their **ToString(…)** method, which first converts them to text and then displays the resulting text content. In fact, the sample code above is equivalent to the following:

```
DateTime currentDate = DateTime.Now;
Console.WriteLine(currentDate.ToString());
```

The default implementation of the **ToString(…)** method in the **object** class returns the full name of the class. All classes that do not explicitly redefine the behavior of the **ToString(…)** are using this implementation. Most classes in C# have their own implementation of the method, which represents readable and understandable content in text form. For example, converting a number to text is using the standard format for numbers in the current

culture. Converting a date to text is also using the standard format for dates in the current culture.

Using of String.Format(…)

`String.Format(…)` is a static method by which we can **format text and other data through a template** (formatting string). The templates contain text and declared parameters (**placeholders**) and are used to obtain formatted text after replacing the parameters with specific values. You can make a direct association with the **`Console.WriteLine(…)`** method, which also formats a string through a template:

```
Console.WriteLine("This is a template from {0}", "David");
```

How to use the **`String.Format(…)`** method? Consider an example in order to clarify this:

```
DateTime date = DateTime.Now;
string name = "David Scott";
string task = "Introduction to C# book";
string location = "his office";

string formattedText = String.Format(
  "Today is {0:MM/dd/yyyy} and {1} is working on {2} in {3}.",
  date, name, task, location);
Console.WriteLine(formattedText);

// Output: Today is 01.02.2012 and David Scott is working on
// Introduction to C# book in his office.
```

As it is seen from the example, formatting with **`String.Format()`** uses **placeholders** (parameters like **`{0}`**, **`{1}`**, etc.) and accepts formatting strings (such as **`:dd.MM.yyyy`**). It accepts as first parameter a formatting string containing text with parameters, followed by values for each parameter and returns the formatted text as a result. More information about formatting strings can be found on the Internet and in the **Composite Formatting** article in MSDN (http://msdn.microsoft.com/en-us/library/txafckwd.aspx). Note that the exact formatting of the output could slightly vary depending on your default culture and internationalization.

Parsing Data

The reverse operation of data formatting is data parsing. **Parsing of data (data parsing)** means to obtain a value of a given type from the text representation of this value in a specific format, i.e. **converting from text to some other data type**, the opposite of **`ToString()`**. For example, from the

text "10/22/2010" we can get an instance of **DateTime** type, containing the relevant date.

Often working with applications with graphical user interface requires the user input to be passed in variables of type **string**. This way we can work well with numbers and characters as well as text and dates, formatted in a user's preferred way. It is up to the developer's experience to represent the expected input data into the right way for the user. The data are then **converted to a specific data type** and processed. For example, numbers can be converted to **int** or **double** variables and then participate in mathematical expressions for calculations.

When converting types, we should not rely only on trusting the user. Always check the correctness of the input user data! Otherwise there could be an exception that could change the normal program logic.

Parsing Numeric Types

To parse a string to a number we can use the **Parse(…)** method of the primitive types. Let's see an example of parsing a string to an integer value:

```
string text = "53";
int intValue = int.Parse(text);
// intValue = 53
```

We can also parse variables of Boolean type:

```
string text = "True";
bool boolValue = bool.Parse(text);
// boolValue = true
```

Return value is **true**, when the passed parameter is initialized (not an object with **null** value), and its content is "**true**" regardless of the casing of letters in it. For example, any text such as "**true**", "**True**" or "**tRUe**" will set the variable **boolValue** to **true**. If the parameter's content is **"false"**, no matter the casing of letters, the return value will be **false**. In all other cases it throws **FormatException**.

In case the passed to the **Parse(…)** method value is invalid for the type (e.g. we pass "John!" when parsing a number), an exception is thrown.

Parsing Dates

Parsing to a date is similar to parsing to a numeric type, but it is recommended to set a specific date format. Here is an example of how this can happen:

```
string text = "11/11/2001";
DateTime parsedDate = DateTime.Parse(text);
Console.WriteLine(parsedDate);
// 11-Nov-01 0:00:00 AM
```

Whether the date will be parsed successfully and in what format exactly it will be printed on the console depends strongly on the current culture of Windows. In the example, a modified version of the U.S. culture (en-US) is used. If we want to set a format explicitly, which does not depend on the culture, we can use the method **DateTime.ParseExact(…)** and specify particular formatting pattern of our choice:

```
string text = "11/12/2001";
string format = "MM/dd/yyyy";
DateTime parsedDate = DateTime.ParseExact(
  text, format, CultureInfo.InvariantCulture);
Console.WriteLine("Day: {0}\nMonth: {1}\nYear: {2}",
  parsedDate.Day, parsedDate.Month, parsedDate.Year);
// Day: 12
// Month: 11
// Year: 2001
```

When parsing with an explicitly set format, it is required to pass a specific **culture** from which to take information about **date format** and **separators** between days and years. Since we want the parsing not to depend on a particular culture, we explicitly specify the neutral culture to be used: **CultureInfo.InvariantCulture**. To use the class **CultureInfo**, we must include the namespace **System.Globalization** in the beginning of our C# source code.

Exercises

1. **Describe the strings in C#.** What is typical for the **string** type? Explain which the most important methods of the string class are.

2. Write a program that reads a string, **reverse** it and prints it to the console. For example: "**introduction**" → "**noitcudortni**".

3. Write a program that **checks whether the parentheses are placed correctly** in an arithmetic expression. Example of expression with correctly placed brackets: **((a+b)/5-d)**. Example of an incorrect expression: **)(a+b))**.

4. How many backslashes you must specify as an argument to the method **Split(…)** in order to **split the text by a backslash**?

   Example: **one\two\three**.

Note: In C# backslash is an escaping character.

5. Write a program that detects how many times a substring is contained in the text. For example, let's look for the substring "**in**" in the text:

```
We are living in a yellow submarine. We don't have anything
else. Inside the submarine is very tight. So we are drinking
all the day. We will move out of it in 5 days.
```

The result is 9 occurrences.

6. A text is given. Write a program that **modifies the casing** of letters to uppercase at all places in the text surrounded by **<upcase>** and **</upcase>** tags. Tags cannot be nested.

Example:

```
We are living in a <upcase>yellow submarine</upcase>. We
don't have <upcase>anything</upcase> else.
```

Result:

```
We are living in a YELLOW SUBMARINE. We don't have ANYTHING
else.
```

7. Write a program that reads a string from the console (20 characters maximum) and if shorter complements it right with "*" to 20 characters.

8. Write a program that converts a given string into the form of array of Unicode escape sequences in the format used in the C# language. Sample input: "**Test**". Result: "**\u0054\u0065\u0073\u0074**".

9. Write a program that **encrypts a text** by applying XOR (excluding or) operation between the given source characters and given cipher code. The encryption should be done by applying XOR between the first letter of the text and the first letter of the code, the second letter of the text and the second letter of the code, etc. until the last letter of the code, then goes back to the first letter of the code and the next letter of the text. Print the result as a series of Unicode escape characters **\xxxx**.

   Sample source text: "**Test**". Sample cipher code: "**ab**". The result should be the following: "**\u0035\u0007\u0012\u0016**".

10. Write a program that **extracts from a text all sentences that contain a particular word**. We accept that the sentences are separated from each other by the character "**.**" and the words are separated from one another by a character which is not a letter. Sample text:

```
We are living in a yellow submarine. We don't have anything
else. Inside the submarine is very tight. So we are drinking
```

```
all the day. We will move out of it in 5 days.
```

Sample result:

```
We are living in a yellow submarine.
We will move out of it in 5 days.
```

11. A string is given, composed of several **"forbidden" words** separated by commas. Also a text is given, containing those words. Write a program that **replaces the forbidden words with asterisks**. Sample text:

```
Microsoft announced its next generation C# compiler today.
It uses advanced parser and special optimizer for the
Microsoft CLR.
```

Sample string containing the forbidden words: "**C#,CLR,Microsoft**".

Sample result:

```
********* announced its next generation ** compiler today.
It uses advanced parser and special optimizer for the
********* ***.
```

12. Write a program that reads a number from console and prints it in **15-character field, aligned right** in several ways: as a decimal number, hexadecimal number, percentage, currency and exponential (scientific) notation.

13. Write a program that **parses an URL** in following format:

```
[protocol]://[server]/[resource]
```

It should **extract** from the URL the protocol, server and resource parts. For example, when **http://www.cnn.com/video** is passed, the result is:

```
[protocol]="http"
[server]="www.cnn.com"
[resource]="/video"
```

14. Write a program that **reverses the words in a given sentence** without changing punctuation and spaces. For example: "**C# is not C++ and PHP is not Delphi**" → "**Delphi not is PHP and C++ not is C#**".

15. A dictionary is given, which consists of several lines of text. Each line consists of a **word and its explanation**, separated by a hyphen:

```
.NET – platform for applications from Microsoft
```

```
CLR – managed execution environment for .NET
namespace – hierarchical organization of classes
```

Write a program that **parses the dictionary** and then reads words from the console in a loop, **gives an explanation** for it or writes a message on the console that the word is not into the dictionary.

16. Write a program that **replaces all hyperlinks** in a HTML document consisting of **…** and hyperlinks in "forum" style, which look like **[URL=…]…[/URL]**.

    Sample text:

```
<p>Please visit <a href="http://softuni.org">our site</a> to
choose a training course. Also visit <a href=
"http://forum.softuni.org">our forum</a> to discuss the
courses.</p>
```

    Sample result:

```
<p>Please visit [URL=http://softuni.org]our site[/URL] to
choose a training course. Also visit [URL=
http://forum.softuni.org]our forum[/URL] to discuss the
courses.</p>
```

17. Write a program that **reads two dates** entered in the format "**day.month.year**" and calculates the **number of days between them**.

```
Enter the first date: 27.02.2006
Enter the second date: 3.03.2006
Distance: 4 days
```

18. Write a program that reads the date and time entered in the format "**day.month.year hour:minutes:seconds**" and prints the date and time after 6 hours and 30 minutes in the same format.

19. Write a program that **extracts all e-mail addresses** from a text. These are all substrings that are limited on both sides by text end or separator between words and match the shape **<sender>@<host>…<domain>**. Sample text:

```
Please contact us by phone (+001 222 222 222) or by email at
example@gmail.com or at test.user@yahoo.co.uk. This is not
email: test@test. This also: @gmail.com. Neither this:
a@a.b.
```

    Extracted e-mail addresses from the sample text:

```
example@gmail.com
test.user@yahoo.co.uk
```

20. Write a program that **extracts from a text all dates** written in format **DD.MM.YYYY** and prints them on the console in the standard format for Canada. Sample text:

```
I was born at 14.06.1980. My sister was born at 3.7.1984. In
5/1999 I graduated my high school. The law says (see section
7.3.12) that we are allowed to do this (section 7.4.2.9).
```

Extracted dates from the sample text:

```
14.06.1980
3.7.1984
```

21. Write a program that extracts from a text all words which are **palindromes**, such as **ABBA**", "**lamal**", "**exe**".

22. Write a program that reads a string from the console and prints in alphabetical order **all letters from the input string and how many times each one of them occurs** in the string.

23. Write a program that reads a string from the console and prints in alphabetical order **all words from the input string** and **how many times each one of them occurs** in the string.

24. Write a program that reads a string from the console and replaces every sequence of identical letters in it with a single letter (the **repeating** letter). Example: "**aaaaabbbbbcdddeeeedssaa**" → "**abcdedsa**".

25. Write a program that reads a list of words separated by commas from the console and prints them in alphabetical order (after **sorting**).

26. Write a program that **extracts all the text without any tags and attribute values** from an HTML document.

Sample text:

```
<html>
  <head><title>News</title></head>
  <body><p><a href="http://softuni.org">Software
    University</a>aims to provide free real-world practical
    training for young people who want to turn into
    skillful software engineers.</p></body>
</html>
```

Sample result:

```
News
Software University aims to provide free real-world
practical training for young people who want to turn into
skillful software engineers.
```

Solutions and Guidelines

1. Read in MSDN or refer to the start of this chapter.
2. Use **StringBuilder** and **for** (or **foreach**) loop.
3. Use **counting of the brackets**: For an opening bracket increase the counter by 1 and for closing bracket decrease it by 1. Watch the counter not to become a negative number and always ends with 0.
4. If you do not know how many slashes you must use, try **Split(…)** with an **increasing number of slashes** until you reach the desired result.
5. Reverse the casing of letters in text to small and **search the given substring in a loop**. Remember to use **IndexOf(…)** with a start index in order to avoid infinite loop.
6. Use **regular expressions** or **IndexOf(…)** method for opening and closing tag. Calculate the start and end index of the text. Change the text in all capital letters and replace the entire substring **opening tag + text + closing tag** with the text in uppercase.
7. Use the **PadRight(…)** method from the **String** class.
8. Use format string "**\u{0:x4}**" for the Unicode character code for each character of the input string (you can get it by converting **char** to **ushort**).
9. Let the cipher **cipher** consists of **cipher.Length** letters. Iterate through all letters in the text and encrypt the letter at position **index** in the text with **cipher[index % cipher.Length]**. If you have a letter from the text and letter from the cipher, we can perform **XOR** operation between them by transforming in advance the two letters into numbers of type **ushort**. We can print the result with "**\u{0:x4}**" format string.
10. First **split the sentences** from each other by using the **Split(…)** method. Then make sure that each sentence **contains the searched word** by searching for it as a substring with **IndexOf(…)** and if you find it check whether there is a separator (character, which is not a letter or start / end of the string) on the left and on the right of the found substring.
11. First, **split the forbidden words** with the method **Split(…)** in order to get them as an array. For each forbidden word, iterate through the text and **search for an occurrence**. If a forbidden word is found, replace it with as many asterisks as letters contained in the forbidden word.

Another, easier approach is to use **RegEx.Replace(…)** with a suitable regular expression and a suitable **MatchEvaluator** method.

12. Use appropriate **formatting strings**.

13. Use a **regular expression** or search for the respective splitters – two slashes for a protocol and one slash as a separator between the server and the resource. Test the special cases like **missing parts of the URL**.

14. You can solve the problem in two steps: **reverse the input string**; **reverse each word in the result string**.

    Another interesting approach is to **split the input text by punctuation marks** between words, in order to get just the words of the text and then **split by the letters** to get the punctuation marks of the text. Thus, given a list of words and a list of punctuation marks between them, you can easily reverse the words, preserving the punctuation marks.

15. You can **parse the text** by splitting it by the new line character, then a second time by the "-" character. The most appropriate way to record the dictionary is in a hash table (**Dictionary<string, string>**), which will provide a quick search for a given word. Read on the Internet for hash-tables and the **Dictionary<K,T>** class. You might also check the chapter "Dictionaries, hash-Tables and Sets".

16. Using a **regular expression** is the easiest way to solve the task.

    If you still choose not to use regular expressions, you can find all substrings that start with "**<a href=**" and end with "****" and within them to replace "****" with "**]**" and then "****" with "**[/URL]**".

17. Use the methods in the **DateTime** structure. For parsing the dates you can use splitting by "**.**" or parsing with the **DateTime.ParseExact(…)** method.

18. Use the **DateTime.ToString()** and **DateTime.ParseExact()** methods with suitable formatting strings.

19. Use **RegEx.Match(…)** with an appropriate **regular expression**.

    If you want to solve the task without regular expressions, you will need to process the text letter by letter from start to finish and process the next character, depending on the current mode, which can be one of **OutsideOfEmail**, **ProcessingSender** or **ProcessingHostOrDomain**. If a separator or the end of the text is reached and host or domain is processed (mode **ProcessingHostOrDomain**), then you have found an e-mail, otherwise potentially a new e-mail is starting and mode must be changed to ProcessingSender. If **@** character is reached in **ProcessingSender** mode, **ProcessingSender** is switched on. When meeting letters or dot in **ProcessingSender** or **ProcessingHostOrDomain** mode, they ate accumulated in a buffer. You can look at all possible

groups of characters encountered respectively in each of the three modes and process them appropriately. We come to something like a final automaton (state machine), which detects e-mail addresses. All found e-mail addresses must be checked whether they have nonempty recipient, host, and domain with a length between 2 and 4 letters, as well as not beginning or ending with a dot.

Another easier approach to this problem is to split the text by all characters that are not letters and dots and to verify that the extracted "words" are valid e-mail addresses. Check can be done through an attempt to split them to nonempty parts: **<sender>**, **<host>**, **<domain>**, meeting the listed conditions.

20. Use **RegEx.Match(…)** with an appropriate **regular expression**. Alternative option is to implement a state-machine that has several states **OutOfDate**, **ProcessingDay**, **ProcessingMonth**, **ProcessingYear** and while processing the text letter by letter to move between states according to the current letter which you are processing. As in the previous task, you can extract all "words" from the text in advance and then check which ones correspond to the date template.

21. Split the text into words and check whether each word is a **palindrome**.

22. Use an array of integers **int[65536]**, which will keep **how many times each letter occurs**. Initially, all array elements are zeros. After processing the input string letter by letter you can write in the array how many times each letter occurs. For example, if you meet the letter '**A**', the number of occurrences in the array index of 65 (Unicode code '**A**') will increase by one. Finally, all non-zero elements (convert array index to **char**, to get the letter) and their number of occurrences can be printed with one scan of the array.

23. Use a hash table (**Dictionary<string, int>**) which keeps how many times each word occurs in the input string. Read on the Internet for class **System.Collections.Generic.Dictionary<K,T>**. With iteration through words you can accumulate information for each word occurrences in the hash table and with hash table iteration you can print the result.

24. You can scan text from left to right and when the current letter is identical with the previous one, miss it, but otherwise concatenate it in **StringBuilder**.

25. Use the static method **Array.Sort(…)** after parsing the input text into array of strings.

26. **Scan the text letter by letter** and at all times keep in a variable whether currently there is an opening tag which has not been closed or not. If you have "**<**", enter in **"opening tag" mode**. If you have "**>**", exit the "opening tag" mode. If you have a letter, add it to the result only if the program is not in "opening tag". After closing a tag you can add a space in order not to "stick" the text before and after the tag.

Chapter 14. Defining Classes

In This Chapter

In this chapter we will understand how to **define custom classes** and their elements. We will learn to declare **fields**, **constructors** and **properties** for the classes. We will revise what a method is and we will broaden our knowledge about **access modifiers** and **methods**. We will observe the characteristics of the constructors and we will set out how the program objects coexist in the dynamic memory and how their fields are initialized. Finally, we will explain what the **static elements** of a class are – fields (including **constants**), properties and methods and how to use them properly. In this chapter we will also introduce generic types (**generics**), enumerated types (**enumerations**) and **nested classes**.

Custom Classes

The aim of every program written by the programmer is to **solve a given problem** based on the implementation of a certain idea. In order to create a solution, first, we sketch a simplified actual model, which does not represent everything, but focuses on these facts, which are significant for the end result. Afterwards, based on the sketched model, we are looking for an answer (i.e. to create an algorithm) for our problem and the solution we describe via given programming language.

Nowadays, the most used programming languages are the object-oriented. And because the **object-oriented programming (OOP)** is close to the way humans think, using one easily allows us to describe models of the surrounding life. Certain reason for this behavior is, because OOP offers tools to draw the set of concepts, which outline classes of objects in every model. The term – class and the definition of custom classes, different from the .NET system framework's, is built-in feature of the C# programming language. The purpose of this chapter is to get us know with it.

Let's Recall: What Does It Mean Class and Object?

Class in the OOP is called a definition (**specification**) of a given type of objects from the real-world. The class represents a pattern, which describes the different states and behavior of the certain objects (the copies), which are created from this class (pattern).

Object is a copy created from the definition (specification) of a given class, also called an **instance**. When one object is created by the description of one class we say **the object is from type "name of the class"**.

For example, if we have a class type **Dog**, which describes some of the characteristics of a real dog, then, the objects based on the description of the class (e.g. the doggies "Fido" and "Rex") are from type class **Dog**. It means the same when the string "some string" is from class type **String**. The difference is that **objects** from type **Dog** is are copies of the class, which is not part of the system library classes of the .NET Framework, but defined by ourselves (the users of the programming language).

What Does a Class Contain?

Every class contains a definition of what kind of data types and objects has in order to be described. The object (the certain copy of this class) holds the actual **data**. The data defines the object's **state**.

In addition to the **state**, in the class is described the **behavior** of the objects. The behavior is represented by actions, which can be performed by the objects themselves. The resource in OOP, through which we can describe this behavior of the objects from a given class, is the declaration of **methods** in the class body.

Elements of the Class

Now, we will go through the main elements of every class, and we will explain them in details latter. The main **elements of a C# classes** are the following:

- **Class declaration** – this is the line where we declare the name of the class, e.g.:

```
public class Dog
```

- **Class body** – similar to the method idioms in the language, the classes also have single class body. It is defined right after the class declaration, enclosed in curly brackets "**{**" and "**}**". The content inside the brackets is known as body of the class. The elements of the class, which are numbered below, are part of the body.

```
public class Dog
{
  // … The body of the class comes here …
}
```

- **Constructor** – it is used for **creating new objects**. Here is a typical constructor:

```
public Dog()
{
  // … Some code …
}
```

- **Fields** – they are variables, declared inside the class (somewhere in the literature are known as **member-variables**). The data of the object, which these variables represent, and are retained into them, is the specific state of an object, and one is required for the proper work of object's methods. The values, which are in the fields, reflect the specific state of the given object, but despite of this there are other types of fields, called **static**, which are shared among all the objects.

```
// Field definition
private string name;
```

- **Properties** – this is the way to describe the **characteristics** of a given class. Usually, the value of the characteristics is kept in the fields of the object. Similar to the fields, the properties may be held by certain object or to be shared among the rest of the objects.

```
// Property definition
private string Name { get; set; }
```

- **Methods** – from the chapter "Methods" we know that methods are named blocks of programming code. They perform particular actions and through them the objects achieve their behavior based on the class type. Methods execute the implemented programming logic (algorithms) and the handling of data.

Sample Class: Dog

Here is how a class looks like. The class **Dog** defined here owns all the elements, which we described so far:

```
// Class declaration
public class Dog
{ // Opening bracket of the class body

   // Field declaration
   private string name;

   // Constructor declaration (peremeterless empty constructor)
   public Dog()
   {
   }

   // Another constructor declaration
   public Dog(string name)
   {
      this.name = name;
```

```
    }

    // Property declaration
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // Method declaration (non-static)
    public void Bark()
    {
        Console.WriteLine("{0} said: Wow-wow!",
            name ?? "[unnamed dog]");
    }
} // Closing bracket of the class body
```

At the moment we will not explain in greater details this code, because the related information will be presented later in this chapter.

Usage of Class and Objects

In the chapter "Creating and Using Objects" we saw in details how new objects of a given class are created and how they can be used. Now, shortly we will revise this programming technique.

How to Use a Class Defined by Us (Custom Class)?

In order to be able to use a given class, first we need to create an object of it. This is done by the reserved word **new** in combination with some of the constructors of the class. This will create an object from a given class (type).

If we want to manipulate the newly created object, we will have to assign it to a variable from its class type. By doing it, in this variable we will keep the connection (reference) to the object.

Using the variable, and the “dot” notation, we can call the methods and the properties of the object, and as well as gain access to the fields (member-variables).

Example – A Dog Meeting

Let’s have the example from the previous section where we defined the class **Dog**, describing a dog, and let’s add a method **Main()** to the class. In this method we will demonstrate how to use the mentioned elements until here: create few **Dog** objects, assign properties to these objects and call methods on these objects:

```
static void Main()
{
   string firstDogName = null;
   Console.Write("Enter first dog name: ");
   firstDogName = Console.ReadLine();

   // Using a constructor to create a dog with specified name
   Dog firstDog = new Dog(firstDogName);

   // Using a constructor to create a dog wit a default name
   Dog secondDog = new Dog();

   Console.Write("Enter second dog name: ");
   string secondDogName = Console.ReadLine();

   // Using property to set the name of the dog
   secondDog.Name = secondDogName;

   // Creating a dog with a default name
   Dog thirdDog = new Dog();

   Dog[] dogs = new Dog[] { firstDog, secondDog, thirdDog };

   foreach (Dog dog in dogs)
   {
      dog.Bark();
   }
}
```

The output from the execution will be the following:

```
Enter first dog name: Axl
Enter second dog name: Bobby
Axl said: Wow-wow!
Bobby said: Wow-wow!
[unnamed dog] said: Wow-wow!
```

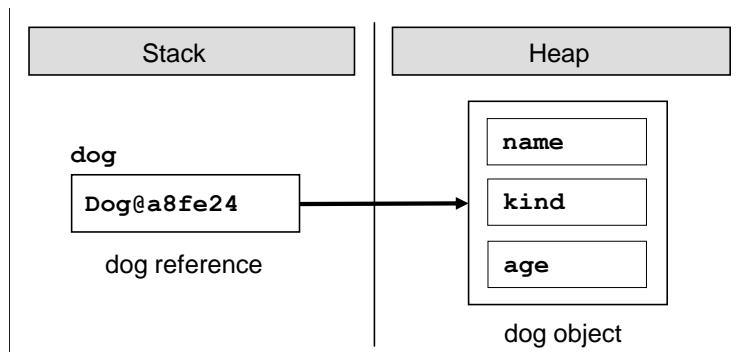
In the example program, with the help of **Console.ReadLine()**, we got the name of the objects of type dog, which the user should input.

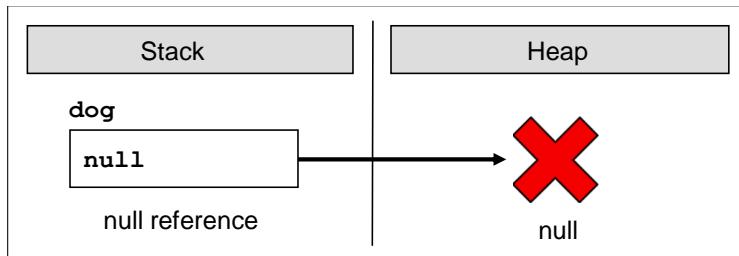
We assigned the first entered string to the variable **firstDogName**. Afterwards we used this variable when we created the first object from class type **Dog** – **firstDog**, by assigning it to the parameter of the constructor.

We created the second object **Dog**, without using a string for the name of the dog in the constructor. With the help of **Console.ReadLine()** we got the

name of the dog and then the value was assigned to the property **Name**. This is done by using a "dot" convention, applied to the variable, which keeps the reference to the second object from type **Dog** – **secondDog.Name**.

When we created the third object from class type **Dog**, we used for the name of the dog its default value which is **null**. Note that in the **Bark()** method dogs whthout name (**name == null**) are printed as "[unnamed dog]".

Afterward we created an array from type **Dog**, by initializing it with the three newly created objects.

At the end, we used a loop, to go through the array of objects from type **Dog.** For every element from the array we again used the "dot" notation, be calling the method **Bark()** for the particular object: **dog.Bark()**.

Nature of Objects

Let's revise, when we create an object in .NET, one consists from two parts – the **significant part (data)**, which contains its data and it is located in the memory of the operating system called a dynamic memory (heap) and a **reference part** to this object, which resides in the other part of the operating system's memory, where are stored the local variable and parameters of the methods (the program execution stack).

For example, let's have a class called **Dog**, which has the properties for name, kind and age. Let's create a variable **dog** from this class. This variable is a reference to the object and is in the dynamic memory (heap).

The **reference** is a variable, which can access objects. The figure below depicts an example reference, which has link to the real object in the heap, and is called with the name **dog**. One, compare to the variable from primitive (value type), does not contain the real value (i.e. the data of the object), but the address, where one is located in the heap memory:

When we declare one variable from type a particular class, and we do not want the variable to be associated with a specific object, then we assign to it the value **null**. The reserved word **null** in the C# language means, that the variable does not point to any object (there is a missing value):

Organizing Classes in Files and Namespaces

In C# the only one limitation regarding the saving of our own custom classes is: they have to be **saved in files with file extension `.cs`**. In such a file several classes, structures and other types can be defined. Although it is not a requirement of the compiler, it is recommended **every class to be stored in exactly one file, which corresponds to its name**, i.e. the class **`Dog`** should be saved in a file **`Dog.cs`**.

Organizing Classes in Namespaces

As we should know from the chapter "Creating and Using Objects", the **namespaces in C# are named group of classes**, which are logically connected, without a requirement how they are stored in the file system.

If we want to include in our code namespaces for the operation in our classes, declared in some file or set of files, this should be done by the so named **`using` directives**. They are not required, but if they exist, they are on the first lines in the class file, before the declaration of the classes or other types. In the next paragraphs we will understand how they exactly are used.

After the insertion of the used namespaces, the next is the declaration of the **namespace** of the classes in the file. As we know, there is no requirement to declare classes in a namespace, but it is a good programming technique if we do it, because the class distribution in the namespace is used for better organization of the code and determination of the classes with equal names.

The namespaces contain classes, structure, interfaces and other types of data, and as well other namespaces. An example of nested namespace is **`System`**, which contains the namespace **`Data`**. The full name of the second namespace is **`System.Data`** and one is nested in the namespace **`System`**.

The **full name of a class** in .NET Framework is the class name, preceded by the namespace in which the class is declared, e.g.: **`<namespace_name>.<class_name>`**. By the **`using`** reserved word we can use types from certain namespace, without writing the full name, e.g.:

```
using System;
…
DateTime date;
```

Instead of:

```
System.DateTime date;
```

One typical declaration sequence, which we should follow when we create custom classes in **.cs** files, is:

```
// Using directives – optional
using <namespace1>;
using <namespace2>;

// Namespace definition - optional
namespace <namespace_name>
{
  // Class declaration
  class <first_class_name>
  {
    // … Class body …
  }

  // Class declaration
  class <second_class_name>
  {
    // … Class body …
  }

  // …

  // Class declaration
  class <n-th_class_name>
  {
    // … Class body …
  }
}
```

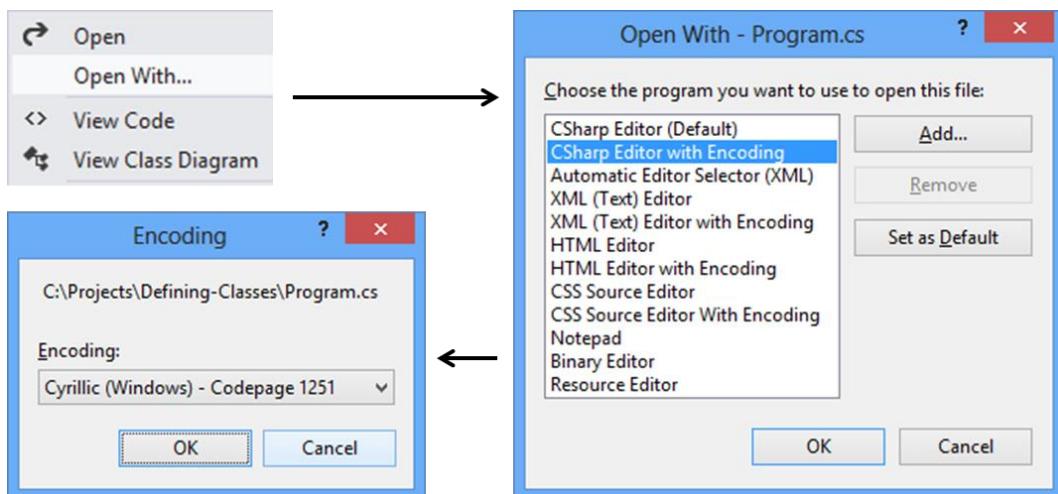
The declaring of the namespace and the relevant include of it is already explained in the chapter "Creating and Using Objects" and therefore we will not discuss it again.

Before we continue, let’s look into the first line of the previous snippet. Instead include of namespace it is a source code comment. This is not a problem in compilation time, the comments are "removed" from the code and thus the first line is still the including statement.

Encoding of Files and Using of Cyrillic and Unicode

While we are creating a **.cs** file, in which to declare our classes, it is good to think about its **character encoding** in the file system.

In the .NET Framework the compiled code is represented in Unicode so it is possible to use characters in our code from alphabets other than Latin. In the next example we use Cyrillic letters for identifiers in Bulgarian language as well as comments in the code, written in Bulgarian (in Cyrillic letters):

```
using System;

public class EncodingTest
{
   // Тестов коментар
   static int години = 4;

   static void Main()
   {
      Console.WriteLine("years: " + години);
   }
}
```

This code will compile and execute without a problem, but to keep the characters readable in the Visual Studio editor we need to provide an appropriate **encoding of the file**.

As we know from the "Strings" chapter, some not all characters can be stored in all encodings. If we use non-standard characters such as Chinese, Cyrillic or Arabic letters, we can use **UTF-8** or other character encoding that supports these characters. By default Visual Studio uses the default character encoding (system locale) defined in the regional settings in Windows. This might be ISO-8859-1 in U.K. or U.S. and Windows-1251 in Bulgaria.

To use a different encoding other than the system's default encoding in Visual Studio, we need to choose the appropriate encoding of the file when opening it in the editor:

1. From the **File** menu we choose **Open** and then **File**.
2. In the **Open File** window we click on the option next to the button **Open** and we choose **Open With...**
3. From the list in the **Open With** window we choose an editor with encoding support, for example **CSharp Editor with Encoding**.
4. Then press **[OK]**.
5. In the window **Encoding** we choose the appropriate encoding from the dropdown menu **Encoding**.
6. Then press **[OK]**.

The steps for saving files in the file system with a specific encoding are:

1. From the **File** menu we choose **Save As**.
2. In the window **Save File As** we press the drop-down box next to the button **Save** and choose **Save with Encoding**.
3. In **Advanced Save Options** we select the desired encoding from the list (preferably the universal UTF-8).
4. From the **Line Endings** we select the desired line ending type.

Although we have the ability to use characters from any non-English alphabet, in **.cs** files it is highly recommended to w**rite all the identifiers and comments in English**, because this way our code will be readable for more people in the world.

Imagine that you live in Germany and you need to type a code written by a Vietnamese person, where the names of all variables and comments are in Vietnamese. You will prefer English, right? Then think about how a developer from Vietnam will handle variables and comments in German.

Modifiers and Access Levels (Visibility)

Let's revise, from the chapter "Methods" we know that a **modifier** is a reserved word and with the help of it we add additional information for the compiler and the code related to the modifier.

In C# there are four **access modifiers**: **public**, **private**, **protected** and **internal**. The access modifiers can be used only in front the following elements of the class: class declaration, fields, properties and methods.

Modifiers and Access Levels

As we explained, in C# there are four access modifiers – **public**, **private**, **protected** and **internal**. Based on them we control the access (visibility) to the elements of the class, in front of which they are used. The levels of access

in .NET are **public**, **protected**, **internal**, **protected internal** and **private**. In this chapter we will review in details only **public**, **private** and **internal**. More about **protected** and **protected internal** we will learn in "Object-Oriented Programming Principles".

Access Level "public"

When we use the modifier **public** in front of some element, we are telling the compiler, that this element **can be accessed from every class**, no matter from the current project (assembly), from the current namespace. The access level **public** defines the miss of restrictions regarding the visibility. This access level is the least restricted access level in C#.

Access Level "private"

The access level **private** is the one, which defines **the most restrictive level of visibility** of the class and its elements. The modifier **private** is used to indicate, that the element, to which is issued, **cannot be accessed from any other class** (except the class, in which it is defined), even if this class exists in the same namespace. This is the default access level, i.e. it is used when there is no access level modifier in front of the respective element of a class (this is true only for elements inside a class).

Access Level "internal"

The modifier **internal** is used to limit the access to the elements of the class only to files **from the same assembly**, i.e. the same project in Visual Studio. When we create several projects in Visual Studio, the classes from will be compiled in different assemblies.

Assembly

.NET assemblies are **collections of compiled types** (classes and other types) and **resources**, which form a logical unit. Assemblies are stored in a binary file of type **.exe** or **.dll**. All types in C# and as general in .NET Framework can reside only inside assemblies. By every compilation of a .NET application one or several assemblies are created by the C# compiler and each assembly is stored inside an **.exe** or **.dll** file.

Declaring Classes

The definition of a class is based on strict syntactical rules:

```
[<access_modifier>] class <class_name>
```

When we declare a class, it is mandatory to use the reserved word **class**. After it must stay the name of the class **<class_name>**.

Besides the reserved word **class** and the name of the class, in the declaration of the class can be used several modifiers, e.g. the reviewed until now modifiers.

Class Visibility

Let's consider two classes – **A** and **B**. We say that, class **A** accesses the elements of class **B**, if the first class can do one of the following:

1. Creates an object (instance) from class type **B**.
2. Can access distinct methods and fields in the class **B**, based on the access level assigned to the particular methods and fields.

There is also another operation, which can be done over the classes, when the visibility allows it. The operation is called **inheritance of a class**, but we will discuss it later in the chapter Object-Oriented Programming Principles.

As we understood, the access level term means "**visibility**". If the class **A** cannot "see" the class **B**, the access level of the methods and the fields in **B** does not matter.

The access levels, which an outer class can have, are only **public** and **internal**. Inner classes can be defined with other access levels.

Access Level "public"

If we declare a class access modifier as **public**, we can reach it from **every class and from every namespace**, regardless of where it exists. It means that every other class can create objects from this type and has access to the methods and the fields of the public class.

Just to know, if we want to use a class with access level **public** from other namespace, different from the current, we should use the reserved word for including different namespaces **using** or every time we should write the full name of the class.

Access Level "internal"

If we declare one class with access modifier **internal**, one will be **accessible only from the same namespace**. It means that only the classes from the same assembly can create objects from this type class and to have access to the methods and fields (with related access level) of the class. This access level is the default, where it is not used access modifier by the declaration of the class.

If we have two projects in common solution in Visual Studio and we want to use a class from one project into the other one then the referenced class should be declared as **public**.

Access Level "private"

If we want to be exhaustive, we have to mention that as access modifier for a class can be used the visibility modifier **private**, but this is related to the term "inner class" (nested class), which we will review in the "Nested Classes" section. Private classes like other private members are accessible only inside the class which defined them.

Body of the Class

By similarity to the methods, after the declaration of the class follows its body, i.e. the part of the class where resides the following programming code:

```
[<access_modifier>] class <class_name>
{
  // … Class body – the code of the class goes here …
}
```

The body of the class begins with opening curly bracket "**{**" and ends with closing one – "**}**". The class always should have a body.

Class Naming Convention

Equal to the methods, for creation of the class names there are the following common standards:

1. The names of the classes begin with capital letter, and the rest of the letters are lower case. If the name of the class consists of several words, every word begins with capital letter, without separator to be used. This is the well-known **PascalCase** convention.
2. For name of the classes **nouns** are usually used.
3. It is recommended the name of the class to be in **English** language.

Here are some example class names, which are following the guidelines:

```
Dog
Account
Car
BufferedReader
```

More about the name of the classes we will learn in the chapter "High-Quality Programming Code".

The Reserved Word "this"

The reserved word **this** in C# is used to **reference the current object**, when one is used from method in the same class. This is the object, which method or constructor is called. The reserved word can be deemed as an

address (reference), given priory from the language authors, with which we access the elements (fields, methods, constructor) of the own class:

```
this.myField; // access a field in the class
this.DoMyMethod(); // access a method in the class
this(3, 4); // access a constructor with two int parameters
```

Currently, we will not explain the given code above. Later, we will do it in other sections of this chapter, dedicated to the elements of the class (fields, methods, constructors) and as well related to the reserved word **this**.

Fields

Objects describe things from the real world. In order to describe an object, we focus on its **characteristics**, which are related to the problems solved in our program. These characteristics of the real-world object we will hold in the declaration of the class in special types of variables. These variables, called **fields** (or member-variables), are holding the **state of the object**. When we create an object based on certain class definition, the values of the fields are containing the characteristics of the created object (its state). These characteristics have different values different for the different objects.

Declaring Fields in a Class

Until now we have discussed only two types of variables (see "Methods") depending on where they are declared:

1. **Local variables** – these are the variables declared in the body of some method (or block).
2. **Parameters** – these are the variables in the list of parameters, which one method can have.

In C# a third type of variable exists, called **field** or **instance variable**.

Fields are declared in the body of the class, outside the body of a single method or constructor.

Fields are declared in the body of the class but not in the bodies of the methods or the constructors.

This is a sample code declaring several fields:

```
class SampleClass
{
  int age;
  long distance;
  string[] names;
  Dog myDog;
```

```
}
```

More formal, the declaration of a field is done in the following way:

```
[<modifiers>] <field_type> <field_name>;
```

The **<field_type>** part determinates the type of a given field. This type can be primitive (**byte**, **short**, **char** and so on), an array, or also some class type (e.g. **Dog** or **string**).

The **<field_name>** part is the name of the field. As the name of the normal variables, when we declare the name of the instance-variables, we should obey the rules for naming of identifiers in C# (see chapter "Primitive Types and Variables").

The **<modifiers>** part is a definition, which describes the access modifiers and as well other modifiers. The last ones are not a mandatory part of the field declaration.

Modifiers and the access modifiers, allowed in the declaration of one field, are explained in chapter "Primitive Types and Variables".

In this chapter, from the other modifiers, which are not based on access levels, and can be used in the declaration of fields, we will discuss **static**, **const** and **readonly**.

Scope

The **scope of a class field** starts from the line where is declared and ends at the closing bracket of the body of the class.

Initialization during Declaration

When we declare one field it is possible to assign to it an initial value. We do this similarly to an assignment of normal local variable:

```
[<modifiers>] <field_type> <field_name> = <initial_value>;
```

Of course, the **<initial_value>** has to be a type compatible with the field's type, e.g.:

```
class SampleClass
{
  int age = 5;
  long distance = 234; // The literal 234 is of integer type

  string[] names = new string[] { "Peter", "Martin" };
  Dog myDog = new Dog();

```

```
    // … Other code …
}
```

Default Values of the Fields

Every time, when we create a new object of a given class, it is allocated memory in the heap for every field from the class. In order this to be done the memory is **initialized automatically with the default values** for the certain field. The fields, which do not have explicitly a default value in the code, use the default value specified for the .NET type, to which they belong.

This is different for the local variables defined in methods. If a local variable in a method does not have a value assigned, the code will not compile. If a member variable (field) in a class does not have a value assigned, it will be automatically zeroed by the compiler.

When an object is created all of the fields are initialized with their respective default values in .NET, except if they are not explicitly initialized with some other value.

In some languages (as C and C++) the newly created objects are not initialized with default values of theirs data and this creates conditions for hard-to-find errors. The last leads to **uncontrolled behavior**, where the program sometimes works correctly (when the allocated memory by chance has good values), and sometimes does not work (when the allocated memory does not contain the proper values). In C# and generally in .NET Framework this problem is solved by the default values for each type coming from the framework.

The value of all types is 0 or something similar. For the most used types these values are as the follows:

Type of the Field	Default Value
bool	false
byte	0
char	'\0'
decimal	0.0M
double	0.0D
float	0.0F
int	0
object reference	null

For more detailed information you can check chapter "Primitive Types and Variables" and its section about the primitive types and their default values.

For example, if we create a class **Dog** and we define for it fields **name**, **age** and **length** and check for the gender **isMale**, without explicitly initializing them, they will be automatically zeroed when we create an object of this class:

```
public class Dog
{
   string name;
   int age;
   int length;
   bool isMale;

   static void Main()
   {
      Dog dog = new Dog();

      Console.WriteLine("Dog's name is: " + dog.name);
      Console.WriteLine("Dog's age is: " + dog.age);
      Console.WriteLine("Dog's length is: " + dog.length);
      Console.WriteLine("Dog is male: " + dog.isMale);
   }
}
```

Respectively, when we execute the program we will have as output the following:

```
Dog's name is:
Dog's age is: 0
Dog's length is: 0
Dog is male: False
```

Automated Initialization of Local Variables and Fields

If we define a local variable in one method, without initializing it, and afterward we try to use it (e.g. printing its value), this will trigger a **compilation error**, because the local variables are not initialized with default values when they are declared.

Unlike fields, local variables are not initialized with default values when they are declared.

Let’s have look into one example:

```
static void Main()
{
   int notInitializedLocalVariable;
   Console.WriteLine(notInitializedLocalVariable);
```

```
}
```

If we try to compile, we will receive the following error:

```
Use of unassigned local variable 'notInitializedLocalVariable'
```

Custom Default Values

A good programming practice is, when we declare fields in the class, to explicitly initialize them with some default value, even if the default value is zero. This will make our code clearer and easy to read.

One example for such initialization is the modified example class **SampleClass** from the previous section:

```
class SampleClass
{
    int age = 0;
    long distance = 0;
    string[] names = null;
    Dog myDog = null;

    // … Other code …
}
```

Modifiers "const" and "readonly"

As was explained in the beginning in this section, in the declaration of one field is allowed to use the modifications **const** and **readonly**. The fields, declared as **const** or **readonly** are called **constants**. They are used when a certain **value is used several times**. These values are declared only ones without repetitions. Examples of constants in the .NET Framework are the mathematical constants **Math.PI** and **Math.E**, and as well the constants **String.Empty** and **Int32.MaxValue**.

Constants Based on "const"

The fields, declared with **const**, have to be initialized during the de facto declaration and afterwards theirs value cannot be changed. They can be accessed without to create an instance (an object) of the class and they are common for all created objects in our program. Something more, when we compile the code, the places where **const** fields are referred are replaced with theirs particular values directly without to use the constant variable at all. For this reason the **const** fields are called **compile-time constants**, because they are replaced with the value during the compilation process.

Constants Based on "readonly"

The modifier **readonly** creates fields, which values cannot be changed once they are assigned. Fields, declared as **readonly**, allow one-time initialization either in the moment of the declaration or in the class constructors. Later theirs values cannot be changed. Because of this reason, the **readonly** fields are called **run-time constants** – constants, because their values cannot be changed after assignment and run-time, because this process happens during the execution of the program (in runtime).

Let's illustrate the foregoing with the following example:

```
public class ConstAndReadOnlyExample
{
    public const double PI = 3.1415926535897932385;
    public readonly double Size;

    public ConstAndReadOnlyExample(int size)
    {
        this.Size = size; // Cannot be further modified!
    }

    static void Main()
    {
        Console.WriteLine(PI);
        Console.WriteLine(ConstAndReadOnlyExample.PI);
        ConstAndReadOnlyExample instance =
            new ConstAndReadOnlyExample(5);
        Console.WriteLine(instance.Size);

        // Compile-time error: cannot access PI like a field
        Console.WriteLine(instance.PI);

        // Compile-time error: Size is instance field (non-static)
        Console.WriteLine(ConstAndReadOnlyExample.Size);

        // Compile-time error: cannot modify a constant
        ConstAndReadOnlyExample.PI = 0;

        // Compile-time error: cannot modify a readonly field
        instance.Size = 0;
    }
}
```

Methods

In chapter "Methods" we have discussed how to **declare and use a method**. In this section we will revise how we do this and we will focus on some additional features from the process of creating methods. Till now we have used static methods only. Now it is time to start using non-static (instance) methods.

Declaring of Class Method

The declaration of methods is done in the following way:

```
// Method definition
[<modifiers>] [<return_type>] <method_name>([<parameters_list>])
{
  // … Method's body …
  [<return_statement>];
}
```

The mandatory elements for declaration of a method are the type of the return value **<return_type>**, the name of the method **<method_name>** and the opening and the closing brackets – "**(**" and "**)**".

The parameter list **<params_list>** is not mandatory. We use it to pass data to the method, which we declare, when this is required.

We know, if the return type **<return_type>** is **void**, then **<return_statement>** can be declared without the **return** statement. If **<return_type>** is different from **void**, the method has to return a result with the help of the reserved word **return** and an expression, which is from the type **<return_type>** or a compatible one.

The work, which the method has to do, is situated in the method body, enclosed in curly brackets – "**{**" and "**}**".

We already discussed some of the access modifiers that can be used in the declaration of a method in the section "Visibility of Methods and Fields" we will review in details this again.

The **static** modifier will be explained in depth in the section "Static Classes and Static Members".

Example – Method Declaration

Let's see the declaration of a method, which sums two values:

```
int Add(int number1, int number2)
{
  int result = number1 + number2;
  return result;
```

```
}
```

The name of the method is **Add** and the return value type is **int**. The parameter list consists of two elements – the variables **number1** and **number2**. Accordingly, the return value is the sum of the two parameters as a result.

Accessing Non-Static Data of the Class

In "Creating and Using Objects", we have discussed how based on the "dot" operator we can access fields and to call the methods of a given class. Now, let's recall how we use conventional non-static methods of a given class, i.e. the methods do not have the modifier **static** in theirs declaration.

E.g. let's have the class **Dog** with the field **age**. To print the value of this field we need to create a **Dog** instance and access the field of this instance via a "dot" notation:

```
public class Dog
{
    int age = 2;

    static void Main()
    {
        Dog dog = new Dog();
        Console.WriteLine("Dog's age is: " + dog.age);
    }
}
```

The result will be:

```
Dog's age is: 2
```

Accessing Non-Static Fields from Non-Static Method

The access to the value of one field can be done via the "dot" notation (as in the last example **dog.age**), or via a method or property. Now, let's create in the class **Dog** a method, which will return the value of **age**:

```
public int GetAge()
{
    return this.age;
}
```

As we see, to access the value of the age field, inside, from the owner class, we use **the reserved word this**. We know that the word **this** is a reference to the current object, in which the method resides. Therefore, in our example,

with "**return this.age**", we say "from the current object (**this**) take (the use of the operator "dot"), the value of the field **age**, and return it as result from the method (with the help of the reserved word **return**). Then, instead from the **Main()** method to access the values of the field **age** of the object **dog**, we simple call the method **GetAge()**:

```
static void Main()
{
  Dog dog = new Dog();
  Console.WriteLine("Dog's age is: " + dog.GetAge());
}
```

The result of the execution based on the change will be the same.

Formally, the declaration of access to a field in the boundaries of a class is the following:

```
this.<field_name>
```

Let's emphasize, that this access option is possible only from non-static code, i.e. method or block, which is without **static** modifier.

Except for retrieving of the value of one field, we can use the reserved word **this** for modification of the field.

E.g., let's declare a method **MakeOlder()**, which will be called every year on the date of the birthday of our pet and this method will increment the age with one year:

```
public void MakeOlder()
{
  this.age++;
}
```

To check if this is correct in the **Main()** method we add the following lines:

```
// One year later, at the birthday date…
dog.MakeOlder();
Console.WriteLine("After one year dog's age is: " + dog.age);
```

After the execution of the program, the result is the following:

```
Dog's age is: 2
After one year dog's age is: 3
```

Calling Non-Static Methods

Like the fields, which do not have **static** modifier in theirs declarations, the methods, which are also non-static, can be called in the body of a class via the reserved word **this**. This is happening again with the "dot" notation and more specifically with the required arguments (if there are any):

```
this.<method_name>(…)
```

For example, let's create a method **PrintAge()**, which prints the age of the object from type **Dog**, and for this purpose calls the method **GetAge()**:

```
public void PrintAge()
{
   int myAge = this.GetAge();
   Console.WriteLine("My age is: " + myAge);
}
```

The first line of the example is indicating that we want to receive the age (the value of the field **age**) of the current object, using the method **GetAge()**. This is done via the reserved word **this**.

The access to the non-static elements of a class (fields and methods) is done via the reserved word this and the operator for access – "dot".

Skip "this" Keyword When Accessing Non-Static Data

When we access the fields of a class or we call its non-static methods, it is possible to **omit the reserved word this**. Then both methods, which we already declared will be written in this way:

```
public int GetAge()
{
   return age; // The same like this.age
}

public void MakeOlder()
{
   age++; // The same like this.age++
}
```

The reserved word **this** is used to indicate **explicitly** that we want to have access to a non-static field of a class or to call some of its non-static methods. When this explicit clarification is not needed, it can be skipped and directly to access the elements of the class.

Although it is understood clearly, the reserved word **this** is often used for access to fields in the class, because it helps to make the code easier to read, understand and maintain, by explicitly stating that we access a field and not a local variable.

When it is not required explicitly the reserved word this can be skipped when we access the elements of the class. For better readability use this keyword even when not required.

Hiding Fields with Local Variables

From the section "Declaring Fields" above, we know that the **scope of one field** starts from the line where the declaration is made to the closing curly bracket of the class. For example let's see the **OverlappingScopeTest** class:

```
public class OverlappingScopeTest
{
    int myValue = 3;

    void PrintMyValue()
    {
        Console.WriteLine("My value is: " + myValue);
    }

    static void Main()
    {
        OverlappingScopeTest instance = new OverlappingScopeTest();
        instance.PrintMyValue();
    }
}
```

This code will have the following result on the console:

```
My value is: 3
```

On the other hand, when we implement the body of one method we have to declare local variables which we will use for the work of the method. As we know, the **scope of a local variable** begins from the line where it is declared to the closing bracket of the body of the method. For example, let’s add this method to the class **OverlappingScopeTest**:

```
Int CalculateNewValue(int newValue)
{
    int result = myValue + newValue;
    return result;
```

```
}
```

In this case, the local variable, which we will use to calculate the new value, is **`result`**.

Sometimes the name of the local variable can overlap with the name of some field. In this case there is a collision.

Let's first look at one example, before we explain what it is about. Let's modify the method **`PrintMyValue()`** in the following way:

```
void PrintMyValue()
{
  int myValue = 5;
  Console.WriteLine("My value is: " + myValue);
}
```

If we declare in this way the method, could it be possible to compile this code? And if it is compiled, is it possible to execute it? If it is compiled and executed which value will be printed – the one of the field or the one of the local variable?

After the execution of the **`Main()`** method, the result will be:

```
My value is: 5
```

This is so, because **C# allows defining local variables, which names match with fields of the class**. If this happens, we say that the scope of the local variable overlays the field variable (**scope overlapping**).

Therefore the scope of the local variable **`myValue`** with value **`5`** overlapped the scope of the field variable in the class. Then, when we print we will get the local variable value.

Despite this, sometimes it is required use the field instead the local variable with the same name. In this case, to retrieve the value of the field, we use the reserved word **`this`**. For this purpose we access the field by using the "dot" operator, applied to the reserved word **`this`**. In this way, we say deliberately that we want to use the field of the class, and not the local variable with the same name.

Let's take a look again at our example relate to the printing of the value **`myValue`**:

```
void PrintMyValue()
{
  int myValue = 5;
  Console.WriteLine("My value is: " + this.myValue);
}
```

This time, after we applied the changes, the result from the call of the method is different:

```
My value is: 3
```

Visibility of Fields and Methods

In the beginning of this chapter we have discussed the generality of the **modifiers and the access levels** for the elements in one class in C#. Later we have discussed the access level in the declaration for one class.

Now we will discuss the **visibility levels of fields and methods** in a class. Because the fields and the methods are elements of the class (members) and have similar rules for access levels, we will expose these rules simultaneously.

Differently from the declaration of a class, when we declare fields and methods in the class we can use the four access levels – **public**, **protected**, **internal** and **private**. The access level **protected** will not be discussed in this chapter, because it is related to class inheritance and is explained in details in the chapter "Object-Oriented Programming Principles".

Before we continue, let's revise, if one class **A** is not visible (does not have access) from other class **B**, then none of its elements (fields and method) can be accessed from class **B**.

If two classes are not visible one to other, then their members (fields and methods) are not visible also, regardless of what kind of access levels their elements have.

In the next subsections, to the explanations until now, we will review examples, in which we have two classes (**Dog** and **Kid**) and which are visible one to other, i.e. every from the classes can create objects from the other type – the other class and to access its elements depending from the defined access level declared. Here is how the first class **Dog** looks like:

```
public class Dog
{
  private string name = "Doggy";

  public string Name
  {
    get { return this.name; }
  }

  public void Bark()
  {
    Console.WriteLine("wow-wow");
```

```
    }

    public void DoSomething()
    {
        this.Bark();
    }
}
```

In addition to the fields and the methods the property **Name** is used, which just returns the field's value. We will discuss in details the property concept later, so currently we will just focus on everything else except the properties.

The code of the class **Kid** looks like this:

```
public class Kid
{
    public void CallTheDog(Dog dog)
    {
        Console.WriteLine("Come, " + dog.Name);
    }

    public void WagTheDog(Dog dog)
    {
        dog.Bark();
    }
}
```

Currently, all elements (fields and methods) of both classes are declared with access modifier **public**, but when we discuss the other access modifiers we will change some of them accordingly. What we would like to find is how the change in the access levels of the elements (fields and methods) of the class **Dog** will be reflected, when the access is made with:

- The own body of the class **Dog**.
- The body of the class **Kid**, respectively, taking into account that **Kid** is in the same namespace (or assembly), in which the **Dog** class is defined or not.

Access Level "public"

When a method or a value of a class is declared with access level **public**, the last **can be used from other classes**, independently from the fact if another class is declared in the same namespace, assembly or outside of it.

Let's review both type of access to members of a class, which are matched in our classes **Dog** and **Kid**:

	The access to the member of the class is done inside the same class **directly** (the class refers itself).
	The access to the member of the class is done via a **reference** to an object created in the body of another class (the class refers another class).

When the members of both classes are **public**, we have the following:

Dog.cs

```
class Dog
{
   public string name = "Doggy";

   public string Name
   {
      get { return this.name; }
   }

   public void Bark()
   {
      Console.WriteLine("wow-wow");
   }

   public void DoSomething()
   {
      this.Bark();
   }
}
```

Kid.cs

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

As we can see, we implement without problem the access to the field **name** and the method **Bark()** of the class **Dog** from the body of the same class. Independently, if the namespace of the class **Kid** is the same as **Dog**, we can, from its body, access the field **name** and to call the method **Bark()** via the "dot" operator, applied to the reference **dog** of the object from type **Dog**.

Access Level "internal"

When a member of some class is declared with access level **internal**, then this element from the class **can be accessed from every class in the same assembly** (i.e. in the same project in Visual Studio), but not from classes outside it (i.e. from other projects in Visual Studio – from the same solution or from a different solution).

Not that if we have a Visual Studio project, all classes in it are from the same assembly and classes defined in different Visual Studio projects (in the same or in a different solution) are from different assemblies.

Below is the explanation about the access level **internal**:

Dog.cs

```
class Dog
{
    internal string name = "Doggy";

    public string Name
    {
        get { return this.name; }
    }

    internal void Bark()
    {
        Console.WriteLine("wow-wow");
    }

    public void DoSomething()
    {
        this.Bark();
    }
}
```

(D) (D)

Respectively, for the class **Kid**, we discuss two cases:

- When the class in **the same assembly**, then the access to the elements of **Dog** will be allowed, independent of whether the classes are in the same namespace or not:

Kid.cs

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

- When the class **Kid** is **external for the assembly**, in which **Dog** is declared, then the access to the field **name** and the method **Bark()** will be denied:

Kid.cs

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }

   public void WagTheDog(Dog dog)
   {
      dog.Bark();
   }
}
```

Actually the access level **internal** for members of the class **Dog** is impossible for two reasons: insufficient visibility of the class and insufficient visibility of its members. To allow access from other assembly to the class **Dog**, one is required to be declared **public** and in the same time its members to be declared as **public**. If the class or its members have lower visibility, the access to it from other assemblies is denied (i.e. from other Visual Studio projects which compile to different **.dll** / **.exe** file).

If we try to compile the class **Kid**, when one is external for the assembly, in which the class **Dog** resides, we will get a compilation error.

Access Level "private"

The access level, which is **the most restrictive**, is **private**. The elements of the class, which are declared with access modifier **private** (or without any, because **private** is the default one), **cannot be accessed outside of the class** in which they are declared.

Therefore, if we declare the field **name** and the method **Bark()** of the class **Dog** with access modifier **private**, there is no problem to access them from the same instance of the class **Dog**, but access from any other classes is not permitted. If you try to access a private method from external class, a compilation error occur. Below is the figure about the access level **private**:

Dog.cs

```
class Dog
{
   private string name = "Doggy";

   public string Name
   {
      get { return this.name; }
   }

   private void Bark()
   {
      Console.WriteLine("wow-wow");
   }

   public void DoSomething()
   {
      this.Bark();
   }
}
```

Accessing the **name** fields from the same class is permitted, but accessing it from a different class (**Kid**) is restricted:

Kid.cs

```
class Kid
{
   public void CallTheDog(Dog dog)
   {
      Console.WriteLine("Come, " + dog.name);
   }
```

```
    public void WagTheDog(Dog dog)
    {
        dog.Bark();
    }
}
```

We should know, when we assign access modifier to a filed, one in most of the cases has to be **private**, because this ensures the highest level of security applied to the field. Respectively, the access and the modification of the value from other classes (if it is required) will be done only via properties or methods. More about this technique we will learn in the section "Properties and Encapsulation of Fields" as well as in the "Encapsulation" section of the chapter "Object-Oriented Programming Principles".

How to Decide Which Access Level to Use?

Before we end up the section regarding visibility of the elements of a class, let's try something. Let's define in the class **Dog** the field **name** and the method **Bark()** witch access modifier **private**. Let's also declare the method **Main()** with the following body:

```
public class Dog
{
    private string name = "Doggy";

    // …

    private void Bark()
    {
        Console.WriteLine("wow-wow");
    }

    // …

    static void Main()
    {
        Dog myDog = new Dog();
        Console.WriteLine("My dog's name is " + myDog.name);
        myDog.Bark();
    }
}
```

The question is, if the class **Dog** can compile when we have declared the elements with access modifier **private** and in the same time is applied a "dot" notation to **myDog** in **Main()**?

The **compilation finished successfully**. Respectively, the result from the execution of the method **Main()** which is declared in the class **Dog** will be the following:

```
My dog's name is Rolf
Wow-wow
```

Everything works, because the access modifiers for the elements of the class are applied to the class and not to a level objects. Because the variable **myDog** is defined in the body of the class **Dog** (where also is situated **Main()** – the start method of the program), we can access its elements (fields and methods) via "dot" notation, regardless we have declared the access level as **private**. If we try to do the same in the body of the class **Kid**, this will be not possible, because the access to **private** fields from outside class is forbidden.

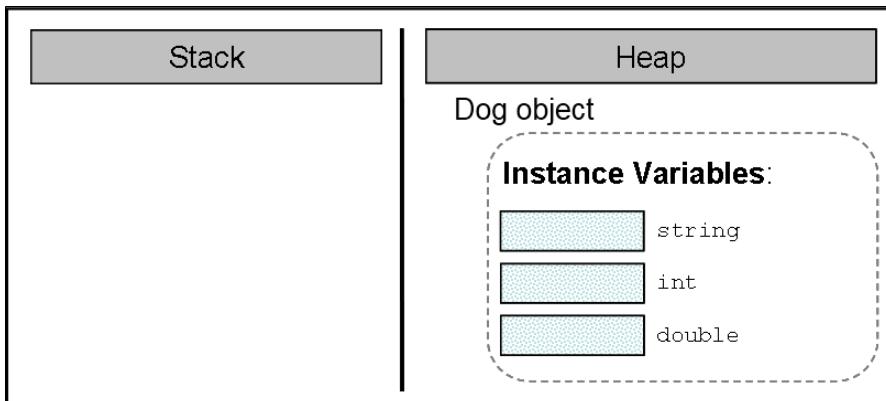
Constructors

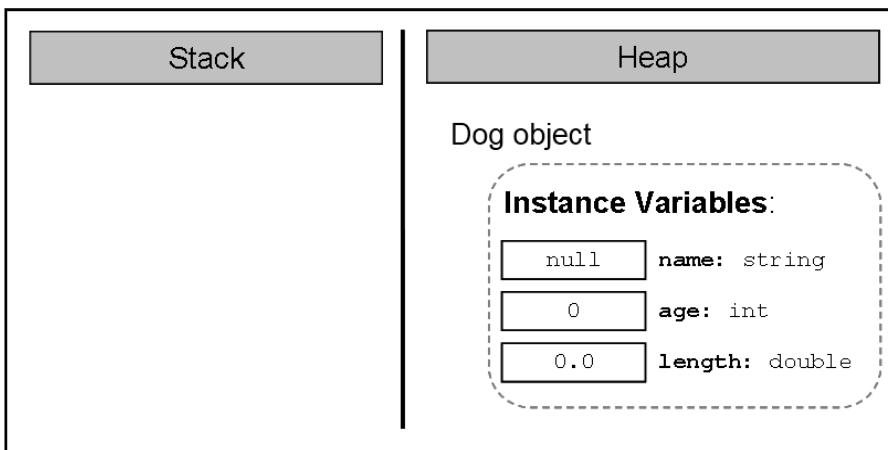
In object-oriented programming, when creating an object from a given class, it is necessary to call a special method of the class known as a **constructor**.

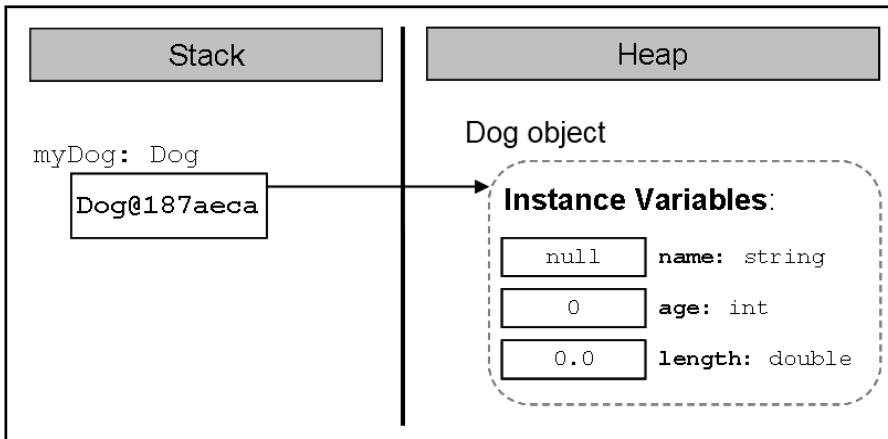
What Is a Constructor?

Constructor of a class is a pseudo-method, which does not have a return type, has the name of the class and is **called using the keyword new**. The task of the constructor is to initialize the memory, allocated for the object, where its fields will be stored (those which are not **static** ones).

Calling a Constructor

The only one way to **call a constructor** in C# is through the **keyword new**. It allocates memory for the new object (in the stack or in the heap, depending on whether the object is a value type or a reference type), resets its fields to zero, calls their constructors (or chain of constructors, formed in succession), and at the end returns a reference to the newly created object.

Consider an example, which will clarify how the constructor works. We know from chapter "Creating and Using Objects" how to create an object:

```
Dog myDog = new Dog();
```

In this case by using the keyword **new** we call the constructor of the class **Dog** and by doing this, memory is allocated, needed for the newly created object of the **Dog** type. When it comes to classes they are allocated in the dynamic memory (in the so called "managed heap").

Let's follow the process of calling a constructor during the creation of new object step by step.

First, **memory is allocated** for the object:

Next, its **fields (if any) are initialized with the default values** for their respective types:

If the creation of the new object is successfully completed, the **constructor returns a reference** to it, which is assigned to the variable **myDog**, from class type **Dog**:

Declaring a Constructor

If we have the class **Dog**, here is how its most simplified constructor (without parameters) will look like:

```
public Dog()
{
}
```

Formally, the declaration of the constructor appears in the following way:

```
[<modifiers>] <class_name>([<parameters_list>])
```

As we already know, the constructors are similar to methods, but they **do not have a return type** (therefore we called them pseudo-methods).

Constructor's Name

In C# it is mandatory that **the name of every constructor matches the name of the class in which it resides – <class_name>**. In the example above the name of the constructor is the same as the name of the class – **Dog**. We should know that, as with methods, the name of the constructor is always followed by round brackets – "**(**" and "**)**".

In (C#) it **is not allowed to declare a method whose name matches the name of the class** (hence the name of the constructors). If nevertheless, a method is declared with the class name, this will cause a compilation error.

```
public class IllegalMethodExample
{
   // Legal constructor
   public IllegalMethodExample ()
   {
   }

   // Illegal method
   private string IllegalMethodExample()
   {
      return "I am illegal method!";
   }
}
```

When we try to compile this class the compiler will display the following **compilation error message**:

```
SampleClass: member names cannot be the same as their enclosing
type
```

Parameter List

Similar to the methods, if we need extra data to create an object, the constructor gets it through a **parameter list** – **<parameters_list>**. In the example constructor of the class **Dog** there is no need of additional data to create an object of this type and therefore there is no parameter list. More about the parameter list will be explained in one of the later sections – "Declaring a Constructor with Parameters".

Of course, after the declaration of the constructor its body is following, which is like every method body in C#, but generally contains mostly initialization logic, i.e. setting the initial values of the fields of the class.

Modifiers

It is evident that **modifiers** can be added in the declaration of the constructors – **<modifiers>**. For modifiers that we know and which are not access modifiers, i.e. **const** and **static**, we should know that only **const** is not allowed to be used in constructors. Later in this chapter, in the "Static Constructors" section we will learn more about the constructors declared with modifier **static**.

Visibility of the Constructors

Similar to the methods and the fields, the constructors can be declared with **levels of visibility**: **public**, **protected**, **internal**, **protected**, **internal** and **private**. The access levels **protected** and **protected internal** will be explained in chapter "Object-Oriented Programming Principles". The rest of the access levels have the same meaning and behavior as with fields and methods.

Initialization of the Fields in the Constructor

As explained earlier when creating a new object and calling its constructor, a new memory is allocated for the non-static fields of the object of the class and they are **initialized with the default values** for their types (see the section "Calling a Constructor").

Furthermore, through the constructors we mainly initialize the fields of the class with values set by us and not with the default ones.

E.g., in the examples we discussed so far, the field **name** of the object from type **Dog** is always initialized during its declaration:

```
string name = "Sharo";
```

Instead of doing this during the declaration of the field, a better programming style is to assign its value in the constructor:

```
public class Dog
```

```
{
   private string name;

   public Dog()
   {
      this.name = "Sharo";
   }

   // … The rest of the class body …
}
```

Although we initialize the fields in the constructor, some people recommend **explicitly assigning their type's default values** during initialization with the purpose of improving the readability of the code, but it is a matter of personal choice:

```
public class Dog
{
   private string name = null;

   public Dog()
   {
      this.name = "Sharo";
   }

   // … The rest of the class body …
}
```

Fields Initialization in the Constructor

Let's see in details what the constructor does after being called and the class fields have been initialized in its body. We know that, when called, it will **allocate memory** for each field and this **memory will be initialized** with the default values.

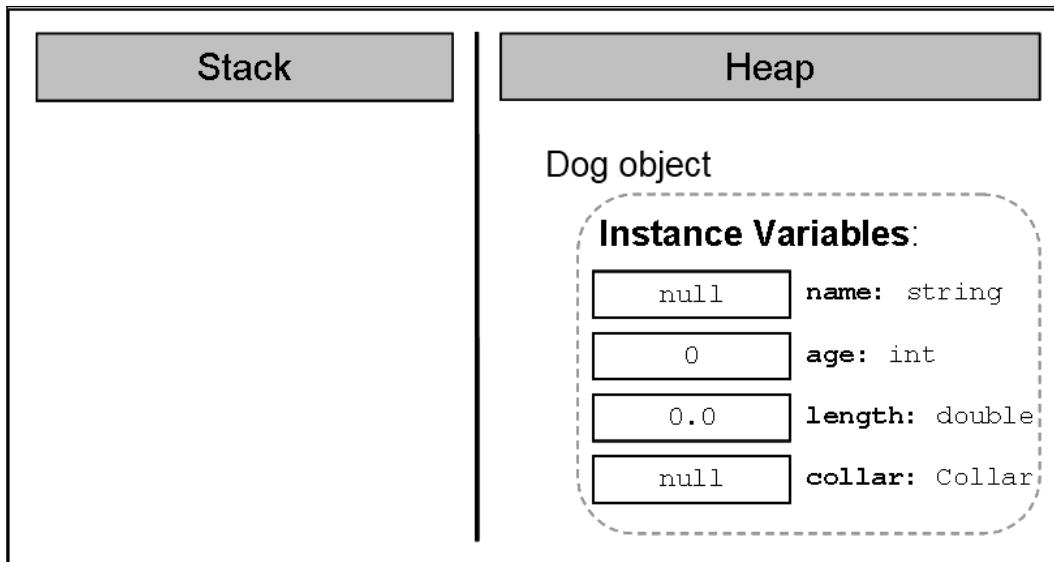
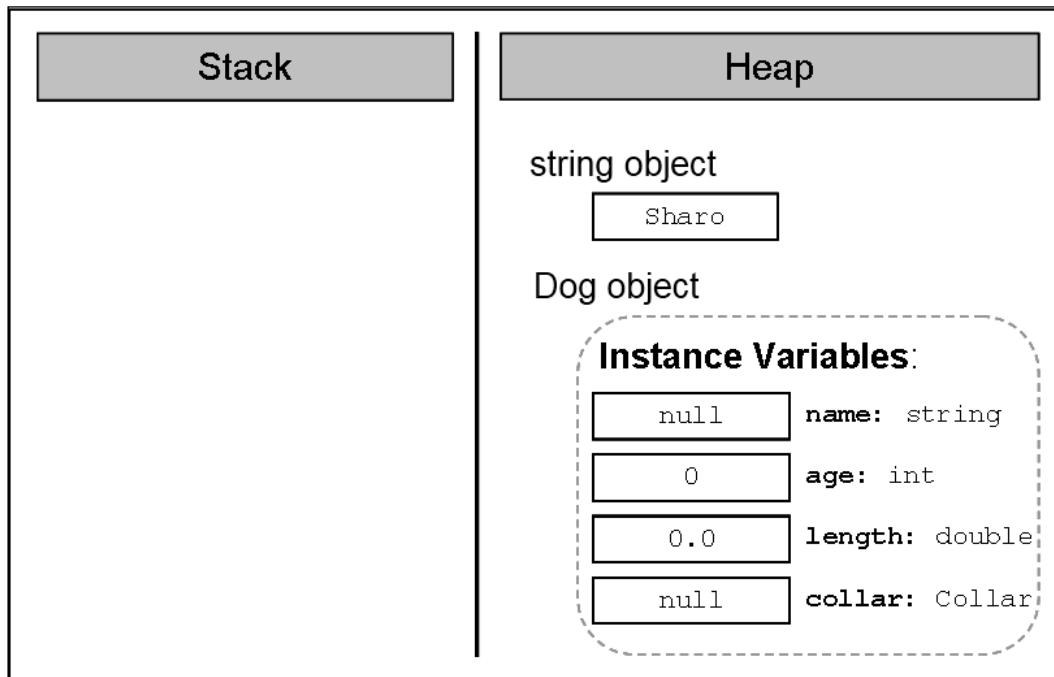
If the fields are of primitive type, then after the default values, we shall assign new values.

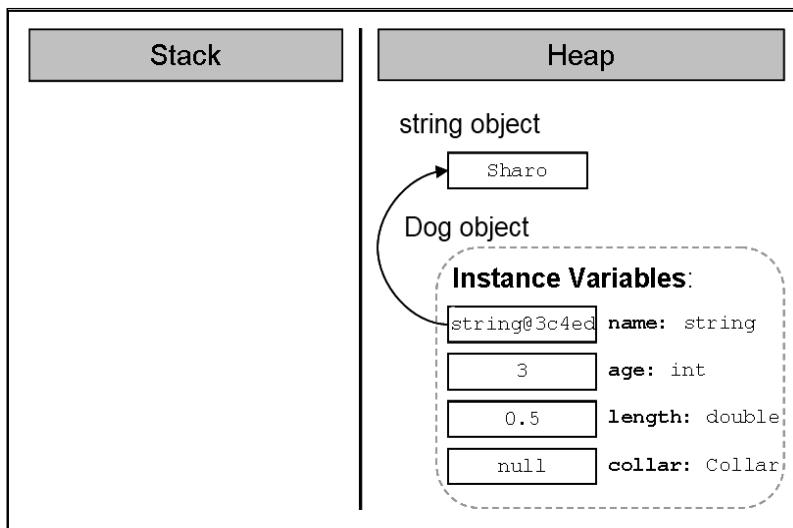
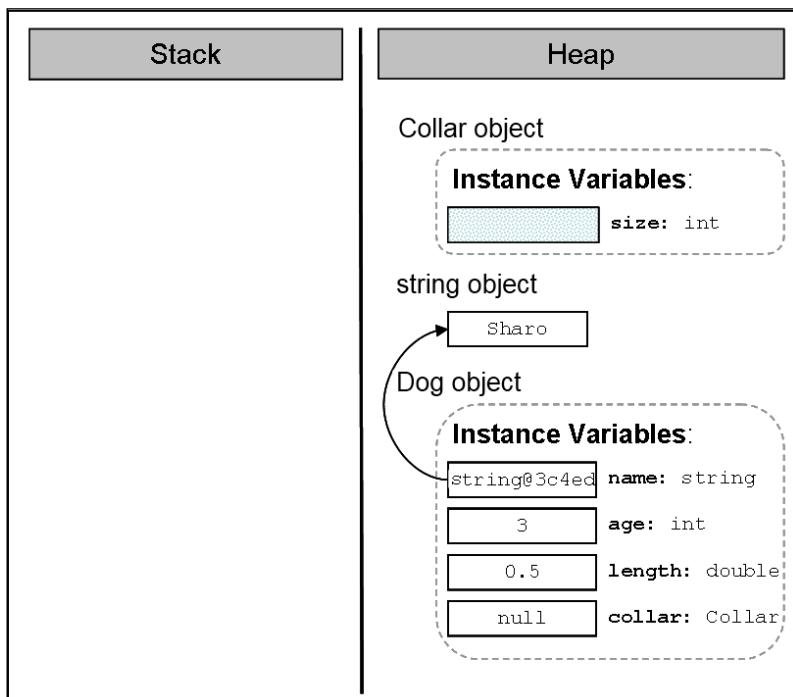
In case the fields are from reference type, such as our field **name**, the constructor will initialize them with **null**. It will then create the object of the corresponding type, in this case the string **"Sharo"** and at the end a reference will be assigned to the new object in the respective field, in our case the field **name**.

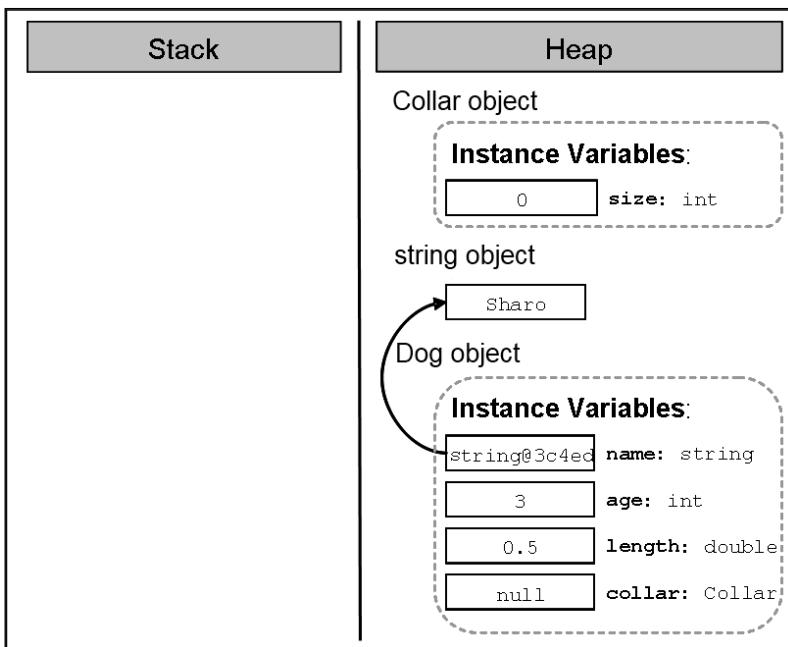
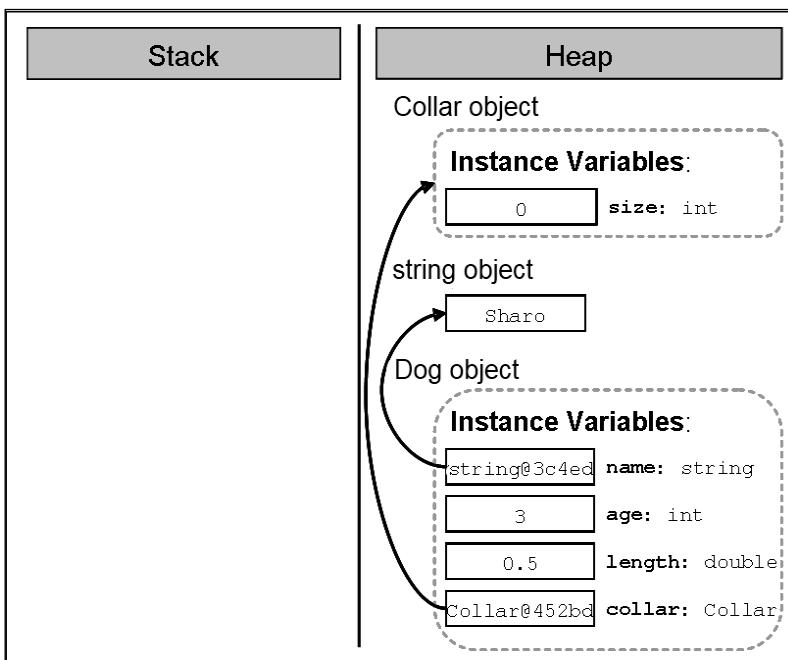
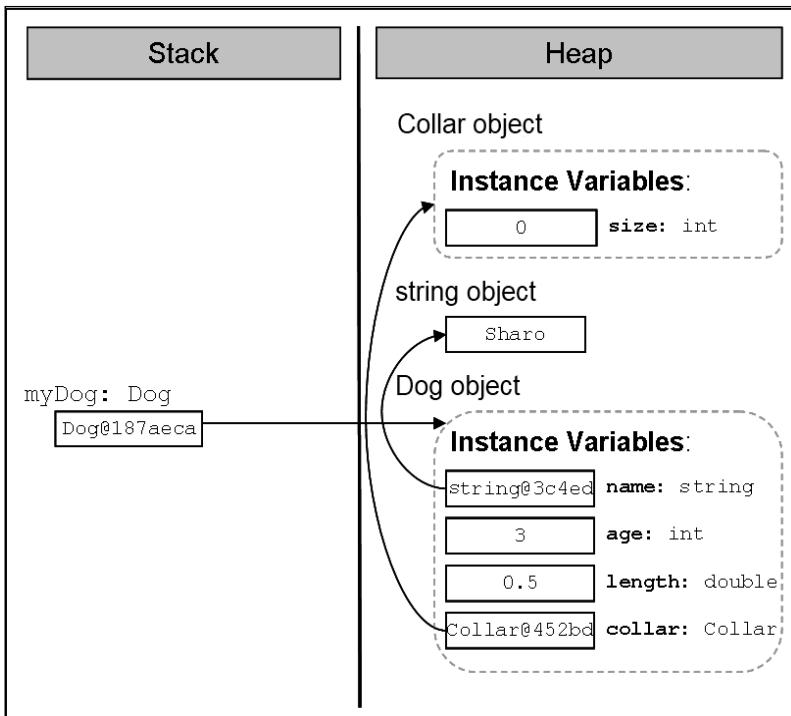
The same will happen if we have other fields, which are not primitive types, and then initialize them in the constructor. E.g. let's have a class called Collar, which describes a dog's accessory – **Collar**:

```
public class Collar
{
  private int size;

  public Collar()
  {
  }
}
```

Let our class **Dog** has a field called **collar**, which is from type **Collar** and which is initialized in the constructor of the class:

```
public class Dog
{
  private string name;
  private int age;
  private double length;
  private Collar collar;

  public Dog()
  {
    this.name = "Sharo";
    this.age = 3;
    this.length = 0.5;
    this.collar = new Collar();
  }

  static void Main()
  {
    Dog myDog = new Dog();
  }
}
```

Representation in the Memory

Let's follow the steps through which the constructor goes, after being called in the **Main()** method.

As we know, as a first step it will **allocate memory in the heap** for all the fields and will initialize them with their default values:

Then, the constructor will have to ensure the creation of the object for the field **name**. It will **call the constructor of the class string**, which will do the work on the string creation):

Stack
Heap
string object
Sharo
Dog object
Instance Variables:
null
name: string
0
age: int
0.0
length: double
null
collar: Collar

Now the constructor will keep the reference to the new string in the field **name** of the **Dog** object:

Then is the creation of the object from type **Collar**. Our constructor (of the class **Dog**) calls the constructor of the class **Collar**, which allocates memory for the object:

Next, the constructor will **initialize it with the default value** for the respective type. The **size** of the **Collar** is not explicitly assigned so it will take the default value for its type (**0** for **int**):

After that the reference to the newly created object, which the constructor of the class **Collar** returns as a result, **will be assigned to the field collar**:

Finally, the reference to the new object from type **Dog will be assigned to the local variable myDog** in the method **Main()**:

Order of Initialization of the Fields

To avoid confusion, let's explain the **order in which the fields of a class are initialized** regardless of whether we have assigned to them values and / or initialized them in the constructor.

First **memory is allocated** for the respective field in the heap and this memory is **initialized** with the default value of the field type. E.g. let's again consider the example with the class **Dog**:

```
public class Dog
{
    private string name;

    public Dog()
    {
        Console.WriteLine(
            "this.name has value of: \"" + this.name + "\"");
        // … No other code here …
    }
    // … Rest of the class body …
}
```

When we try to create a new object of our class type the console will show:

```
this.name has value of: ""
```

After the initialization of the fields with the default value for the respective type, the second step in CLR (Common Language Runtime) is to **assign a value to the field** if such has been set when declaring the field.

So, if we change the line in the class **Dog**, where we declare the field **name**, it will first be initialized with the value **null** and then it will be assigned the value "**Rex**".

```
private string name = "Rex";
```

Respectively, for every creation of a new object of the class:

```
static void Main()
{
   Dog dog = new Dog();
}
```

The following will be printed:

```
this.name has value of: "Rex"
```

Only after these two steps of initializing the fields of the class (default value initialization and possibly the value set by the programmer during the declaration of the field) **the constructor of the class is called**. At this time, the fields get the values, which are set in the body of the constructor.

Declaring a Constructor with Parameters

In the previous section, we saw how we can set values to the fields, other than the default values. Very often, however, during the declaration of the constructor, we don't know what values the various fields will take. To tackle this problem, the required information, **similar to the methods with parameters**, the fields are assigned the values, given to them in the body of the constructor. For example:

```
public Dog(string dogName, int dogAge, double dogLength)
{
   name = dogName;
   age = dogAge;
   length = dogLength;
   collar = new Collar();
}
```

Similarly, the **call of a constructor with parameters** is done in the same way as the call of method with parameters – the required values are supplied as a list, the elements of which are separated with commas:

```
static void Main()
{
  Dog myDog = new Dog("Moby", 2, 0.4); // Passing parameters

  Console.WriteLine("My dog " + myDog.name +
    " is " + myDog.age + " year(s) old. " +
    " and it has length: " + myDog.length + " m.");
}
```

The result of the execution of this **Main()** method is the following:

```
My dog Moby is 2 year(s) old. It has length: 0.4 m.
```

There is no limitation for the number of the constructors of a class in C#. The only requirement is that they **differ in their signature** (what signature is we already explained in chapter "Methods").

Scope of Parameters of the Constructor

By analogy with the scope of the variables in the parameter list of a method, the **variables in the parameter list of one constructor have a scope** from the opening bracket of the constructor to the closing bracket, i.e. throughout the body of the constructor.

Very often, when we declare a constructor with parameters it is possible to name the variables from the parameter list with **the same names** as the names of the fields, which are going to be initialized. Let's, for example, consider the constructor of the class **Dog**:

```
public Dog(string name, int age, double length)
{
  name = name;
  age = age;
  length = length;
  collar = new Collar();
}
```

Let's compile and execute the **Main()** method declared a little bit above:

```
My dog is 0 year(s) old. It has length: 0 m
```

Strange result, isn't it? In fact this result is not so awkward. The explanation is the following: the scope, in which the variables from the list of the constructor parameters are acting, overlaps the scope of acting of the fields

with the same names in the constructor. Thus, **we do not assign any value to the fields** because in practice we have no access to them. For example, instead of assigning the variable value to the field **age**, we assign the value of the variable **age** to the variable itself:

```
age = age;
```

As we saw from the section "Hiding Fields with Local Variables", to avoid this problem we should access the field, to which we want to assign a value, **using the keyword this**:

```
public Dog(string name, int age, double length)
{
  this.name = name;
  this.age = age;
  this.length = length;
  this.collar = new Collar();
}
```

Now, assuming we execute again the **Main()** method:

```
static void Main()
{
  Dog myDog = new Dog("Moby", 2, 0.4);

  Console.WriteLine("My dog " + myDog.name +
     " is " + myDog.age + " year(s) old. " +
     " and it has length: " + myDog.length + " m");
}
```

The result will be exactly what we expect it to be:

```
My dog Moby is 2 year(s) old. It has length: 0.4 m
```

Constructor with Variable Number of Arguments

Similar to methods with **variable number of arguments**, discussed in chapter "Methods", constructors can also be declared with a parameter for a variable number of arguments. The rules for declaring and calling constructors with a variable number of arguments are the same as the ones, described for declaring and calling with the methods:

1. When we declare a constructor with variable number of arguments, we must use **the reserved word params**, and then insert the type of the parameters, followed by square parentheses. Finally the name of the array follows, in which array the arguments used for the calling of the

method are stored. For example for whole number arguments we can use **params int[] numbers**.

2. It is allowed for the constructor with a variable number of arguments to have other parameters too in the parameter list.
3. The parameter for the variable number of arguments must be the last in the parameter list of the constructor.

Consider a **sample declaration** of a constructor of a class, which describes a lecture:

```
public Lecture(string subject, params string[] studentsNames)
{
  // … Initialization of the instance variables …
}
```

The first parameter in the declaration is the name of the subject of the lecture and the next parameter represents a **variable number of arguments** – the names of the students. Here is how a sample object of this class would be constructed:

```
Lecture lecture =
  new Lecture("Biology", "Peter", "Mike", "Steven");
```

Accordingly, as the first parameter is the name of the subject – **"Biology"**, and all the rest arguments – the names of the attending students.

Constructor Overloading

As we saw, we can declare constructors with parameters. This gives us a possibility to declare constructors with different signatures (number and order of the parameters) with the purpose of providing convenience to those who will create objects from our class. Creating **constructors with different signatures** is called **constructor overloading**.

Consider, for example, the class **Dog**. We can declare different constructors:

```
// No parameters
public Dog()
{
  this.name = "Axl";
  this.age = 1;
  this.length = 0.3;
  this.collar = new Collar();
}

// One parameter
```

```
public Dog(string name)
{
    this.name = name;
    this.age = 1;
    this.length = 0.3;
    this.collar = new Collar();
}

// Two parameters
public Dog(string name, int age)
{
    this.name = name;
    this.age = age;
    this.length = 0.3;
    this.collar = new Collar();
}

// Three parameters
public Dog(string name, int age, double length)
{
    this.name = name;
    this.age = age;
    this.length = length;
    this.collar = new Collar();
}

// Four parameters
public Dog(string name, int age, double length, Collar collar)
{
    this.name = name;
    this.age = age;
    this.length = length;
    this.collar = collar;
}
```

Reusing Constructors

In our last example we saw that, depending on the needs for creating objects of our class, we can declare different variants of the constructors. It is easy to notice that a large part of the **constructor code is repeated**. This leads us to the question whether there is an alternative way for a constructor, which is already doing an initializing, to be reused by the others to perform the same initialization. On the other hand, at the beginning of the chapter it was mentioned that a constructor cannot be called in the manner in which the

methods are called but by the keyword **new**. There should be a way – otherwise a lot of code will be repeated unnecessarily.

In C# a mechanism exists through which **one constructor can call another** one declared in the same class. This is done again with the keyword **this**, but used in another syntax structure in declaring the constructors:

```
[<modifiers>] <class_name>([<parameters_list_1>])
   : this([<parameters_list_2>])
```

To the well-known form of declaring a constructor (the first line of the declaration above), we can add a colon, followed by the keyword **this**, followed by parentheses. If the constructor we want to call has parameters, in the brackets we need to add a list of parameters **parameters_list_2** to be supplied.

Here is how the code from the section about constructor overloading would look like, in which instead of repeating the initialization of each of the fields, we will call the constructors declared in the same class:

```
// No parameters
public Dog()
   : this("Axl") // Constructor call
{
   // More code could be added here
}

// One parameter
public Dog(string name)
   : this(name, 1) // Constructor call
{
}

// Two parameters
public Dog(string name, int age)
   : this(name, age, 0.3) // Constructor call
{
}

// Three parameters
public Dog(string name, int age, double length)
   : this(name, age, length, new Collar()) // Constructor call
{
}

// Four parameters
```

```
public Dog(string name, int age, double length, Collar collar)
{
  this.name = name;
  this.age = age;
  this.length = length;
  this.collar = collar;
}
```

As indicated by comments in the first constructor in the example above, if necessary, in addition to calling any of the other constructors with certain parameters, every constructor can add into its body a code, which performs additional initializations or other actions.

Default Constructor

Consider the following question – what happens if we don't declare a constructor in our class? How can we create objects from this type?

As it often happens, when a class is without a single constructor, this issue is resolved by C#. When we do not declare any constructors, the compiler will create one for us and this one will be used to create objects such as our class. This constructor is called **default implicit constructor** and it will not have any parameters and will be empty (i.e. it will not do anything in addition to the default zeroing of the object fields).

When we do not declare any constructor in a given class, the compiler will create one, known as a default implicit constructor.

For example, let's declare the class **Collar**, without declaring any constructor in it:

```
public class Collar
{
  private int size;

  public int Size
  {
    get { return size; }
  }
}
```

Although we do not have an explicitly declared constructor without parameters, we can create objects of this class in the following way:

```
Collar collar = new Collar();
```

The **default parameterless constructor** looks the following way:

```
<access_level> <class_name>() { }
```

We should know that the default constructor is always named like the class **<class_name>**, and its parameter list is always empty as well as its body. The compiler simply adds one if there is no constructor in the class. The default constructor is usually **public** (except for some very specific situations, where it is **protected**).

The default constructor is always without parameters.

To make sure that the default constructor is always without parameters let's try to call the default constructor by setting it with parameters:

```
Collar collar = new Collar(5);
```

The compiler will display the following error message:

```
'Collar' does not contain a constructor that takes 1 arguments
```

How the Default Constructor Works?

As we can guess, the only thing the default constructor will do when creating objects of our class, is to zero the fields of the class. For example, if in the class **Collar** we have not declared any constructor and we create an object from it, and later we try to print the value in the field **size**:

```
static void Main()
{
  Collar collar = new Collar();
  Console.WriteLine("Collar's size is: " + collar.Size);
}
```

The result will be:

```
Collar's size is: 0
```

We see that the value saved in the field **size** of the object **collar** is just the default value of the whole number type – **int**.

When a Default Constructor Will Not Be Created?

We have to know that if we declare at least one constructor in a given class then the compiler will not create a default constructor.

To investigate this, consider the following example:

```
public Collar(int size)
  : this()
{
  this.size = size;
}
```

Let this be **the only constructor in the class `Collar`**. We try to call a constructor without parameters in it, hoping that the compiler will have created a default parameterless constructor for us. After we try to compile, we will find out that what we are trying to do is not possible. The compiler will show the following error:

```
'Collar' does not contain a constructor that takes 0 arguments
```

The rule about the default implicit parameterless constructor is:

If we declare at least one constructor in a given class, the compiler will not create a default constructor for us.

Difference between a Default Constructor and a Constructor without Parameters

Before we finish this section for the constructors, we will clarify something very important:

Although the default constructor and the one without parameters are similar in signature, they are completely different.

The difference is that the default implicit constructor is created by the compiler, if we do not declare any constructor in our class, and the **constructor without parameters** is declared by us.

Moreover, as explained earlier, the default constructor will always have access level **`protected`** or **`public`**, depending on the access modifier of the class, while the level of access of the constructor without parameters all depends on us – we define it.

Properties

In the world of object-oriented programming there is an element of the classes called **property**, which is **somewhere between a field and a method** and serves to better protect the state in the class. In some languages for object-oriented programming, like C#, Delphi / Pascal, Visual Basic, Python, JavaScript, and others, the properties are a part of the language, i.e. there is a special mechanism to declare and use them. Other languages like Java do not support the property concept and for this purpose

the programmers should declare a pair of methods (for reading and modifying the property) to provide this functionality.

Properties in C# – Introduction by Example

Using the properties is a good and proven practice and an important part of the concepts for object-oriented programming. The creation of a property in programming is done by **declaring two methods** – one for access (**reading**) and one for modifying (**setting**) the value of the respective property.

Consider an example. Assume we have again class **Dog**, which describes a dog. A characteristic of a dog is, for example, its color. The access to the property "color" of a dog and its corresponding modification can be accomplished in the following way:

```
// Getting (reading) a property
string colorName = dogInstance.Color;

// Setting (modifying) a property
dogInstance.Color = "black";
```

Properties – Encapsulation of Fields

The main objective of the properties is to ensure the **encapsulation of the state of the class** in which they are declared, i.e. to protect the class from falling into **invalid state**.

Encapsulation is **hiding of the physical representation** of data in one class so that if we subsequently change this presentation, it will not reflect on other classes, which use this class.

Though the C# syntax this is done by declaring the fields (physical presentation of data) with possibly the most limited level of visibility (mostly with the modifier **private**) and declaring that access to these fields (reading and modifying) is to take place only through special **accessor methods**.

Example of Encapsulation

To illustrate what the encapsulation, which provides properties to a class, is and what the properties themselves represent, we shall consider an example.

Let's have a class, which represents a **point from the 2D space** with properties representing the coordinates {x, y}. Here is how it would look like if we declare each of the coordinates as a field:

Point.cs

```
class Point
{
```

```
    private double x;
    private double y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public double X
    {
        get { return this.x; }
        set { this.x = value; }
    }

    public double Y
    {
        get { return this.y; }
        set { this.y = value; }
    }
}
```

The fields of the objects of our class (i.e. the point's coordinates) are declared as **private** and cannot be accessed by a "dot" notation. If we create an object from class **Point**, we can modify and read the properties (the coordinates) of the point only through the properties for access to them:

PointTest.cs

```
using System;

class PointTest
{
    static void Main()
    {
        Point myPoint = new Point(2, 3);

        double myPointXCoord = myPoint.X; // Access a property
        double myPointYCoord = myPoint.Y; // Access a property

        Console.WriteLine("The X coordinate is: " + myPointXCoord);
        Console.WriteLine("The Y coordinate is: " + myPointYCoord);
    }
}
```

The result of the execution of the **Main()** method will be:

```
The X coordinate is: 2
The Y coordinate is: 3
```

If, however, we decide to change the internal representation of the point's properties, e.g. instead of two fields, we declare them as a one-dimensional array with two elements; we can do it without affecting in any way of the other classes of our project:

Point.cs

```
using System;

class Point
{
  private double[] coordinates;

  public Point(int xCoord, int yCoord)
  {
    this.coordinates = new double[2];

    // Initializing the x coordinate
    coordinates[0] = xCoord;

    // Initializing the y coordinate
    coordinates[1] = yCoord;
  }

  public double X
  {
    get { return coordinates[0]; }
    set { coordinates[0] = value; }
  }

  public double Y
  {
    get { return coordinates[1]; }
    set { coordinates[1] = value; }
  }
}
```

The result of the implementation of the **Main()** method will not be changed and will be the same even without changing a single character in the code of the class **PointTest**.

The demonstration is a **good example of data encapsulation** of an object, provided by the mechanism of the properties. Through them we **hide the internal representation** of the information by declaring properties and methods for accessing it, and if later a change occurs in the representation, this will not affect the other classes using our class, because they only use its properties and do not know how the information is represented "behind the scene".

Of course, the example shows only one of the benefits of class fields wrapping (packing) into properties. **Properties allow further control over the data** in the class and they can check whether the assigned values are correct according to some criteria. For example, if we have a property "maximum speed" for a class **Car**, it is possible, through properties, to require its value to be within the range of 1 to 300 km/h.

Physical Presentation of the Properties in a Class

As we saw above, the properties may have **different presentation in one class** at a physical level. In our example, the information about the properties of the class **Point** initially was stored in two fields and later in one field–array.

However, if we decide instead of keeping the information about the properties of the point in a field, to save it in a file or a database and every time we need to access the respective property, we can read or write from the file or the database rather than use the fields of the class as in the previous examples. Since the properties are accessed by special methods (called methods for access and modification or **accessor methods**) to be discussed later, for the classes that will use our class the question how the information will be stored would not matter (because of the good encapsulation).

In the most common case, however, the information about the properties of the class is saved in a field of the class, which has the most rigorous level of visibility – **private**.

It does not matter how the information for the properties in a class in C# is saved, but usually this is done by a class field with the most restrictive access level (private).

Property without Declaration of a Field

Consider an example, in which the property is stored neither in the field, nor anywhere else, but recalculated when trying to access it.

Let's have the class **Rectangle**, which represents the geometric shape of a rectangle. Accordingly, this class has two fields – for **width** and for **height**. Assume our class has one more property – **area**. Because we always can **calculate the property "area"** of a rectangle based on the width and the height, it is not required to define a separate field in the class to keep this value. Therefore, we can simply declare a method for obtaining the area through which we calculate the area of a rectangle:

Rectangle.cs

```
using System;

class Rectangle
{
  private float height;
  private float width;

  public Rectangle(float height, float width)
  {
    this.height = height;
    this.width = width;
  }

  // Obtaining the value of the property area
  public float Area
  {
    get { return this.height * this.width; }
  }
}
```

As we will see later, a property does not necessarily have an accessing and a modifying method at the same time. Therefore, it is allowed to declare only a method for reading the property **Area** of the rectangle. There is no point to have a method, which modifies the value of the area of a rectangle because the area is always one and the same based on given lengths of the sides.

Declaring Properties in C#

To declare a property in C#, we have to declare access methods (for reading and changing) of the respective property and to decide how we will store the information related to the property in the class.

Before we declare the methods, however, we have to declare the property of the class. Formal declaration of properties appears in the following way:

```
[<modifiers>] <property_type> <property_name>
```

With **<modifiers>** we have denoted both the **access modifiers and other modifiers** (e.g. **static**, to be discussed in the next section of this chapter). They are not a mandatory part of the declaration of a field.

The **type of the property <property_type>** specifies the type of the values of the property. It may be either a primitive type (e.g. **int**), or a reference type (e.g. array).

Respectively, **<property_name>** is **the name of the property**. It must begin with a capital letter and to satisfy the **PascalCase** rule, i.e. every new word that is adjoined to the end part of the property name, starts with a capital letter. Here are some examples of properly named properties:

```
// MyValue property
public int MyValue { get; set; }

// Color property
public string Color { get; set; }

// X-coordinate property
public double X { get; set; }
```

The Body of a Property

Like classes and methods in C# properties also have **bodies**, where the methods for access are declared (**accessors**).

```
[<modifiers>] <property_type> <property_name>
{
  // … Property's accessors methods go here
}
```

The body of a property begins with an opening bracket "**{**" and ends with a closing bracket – "**}**". Properties should always have a body.

Method for Reading the Value of a Property (Getter)

As we explained, the declaration of a **method for reading a value of a property** (in the literature called a **getter**) is made in the body of a property by using the following syntaxes:

```
get { <accessor_body> }
```

The content of the block surrounded by the braces (**<accessor_body>**) is similar to the contents of any method. The actions, which should be performed to return the result of the method, are declared in it.

The method of reading the value of a property must end with a **return** or **throw** operation. The type of the value, which is returned as a result of this method, has to be the same as **<property_type>** described in the property declaration.

Although earlier in this section we considered many examples of declared properties with a method for reading their values, let's consider another example of a property – **Age**, which is of type **int** and is declared via a field in the same class:

```
private int age;            // Field declaration

public int Age          // Property declaration
{
  get { return this.age; }  // Getter declaration
}
```

Calling a Method for Reading Property's Value

Assume that the property **Age** from the last example is declared in the class **Dog**. Then calling the method for reading the value of the property is done by a "dot" notation, applied to a variable of the type, in the class of which the property is declared:

```
Dog dogInstance = new Dog();
// …
int dogAge = dogInstance.Age;          // Getter invocation
Console.WriteLine(dogInstance.Age);   // Getter invocation
```

The last two lines of the example show that when accessing through a dot notation the name of the property, its getter method (method for reading its value) is called automatically.

Method for Modifying Property's Value (Setter)

Like the method of reading the property's value we can also declare the method of changing (**modifying) the value of a property** (in the literature known as **setter**). It is declared in the body of a property with **void** return value and the assigned value is accessible through an implicit parameter **value**.

The declaration is made in the body of the property through the following syntax:

```
set { <accessor_body> }
```

The contents of the block surrounded by arrow brackets – **<accessor_body>** are similar to the content of any method. It declares the actions that must be performed to change the value of the property. The method uses a hidden parameter called **value**, which is available in C# by default and contains the new value of the property. The type of the parameter is the same as the type of the property.

Let's add the example for the property **Age** in the class **Dog** to illustrate what we discussed so far:

```
private int age;            // Field declaration
```

```
public int Age          // Property declaration
{
  get { return this.age; }
  set { this.age = value; } // Setter declaration
}
```

Calling a Method for Modifying the Property's Value

Calling the method to modify the property's value is performed via the "dot" notation, applied to the variable of the type, in the class of which the property is declared:

```
Dog dogInstance = new Dog();
// …
dogInstance.Age = 3;          // Setter invocation
```

In the last line where the value 3 is assigned the setter method of the property **Age** is called. In this way the value is saved in the parameter **value** and is assigned to the setter method of the property **Age**. In our example, the value of the variable **value** is assigned to the field **age** from the class **Dog**, but in the general case this can be handled in a more complicated way.

Assertion of the Input Values

It is a good practice in the programming process to **check the validity of the input values** for the setter method of modifying a property and if they are not valid to take the necessary "measures". Mostly, in case of incorrect input data an exception is caused.

Consider again the example with the age of the dog. As we know the age has to be a positive number. To prevent someone from assigning a negative number or a zero to the property **Age**, we add the following validation at the beginning of the setter method:

```
public int Age
{
  get { return this.age; }
  set
  {
    // Take precaution: perform check for correctness
    if (value < 0)
    {
      throw new ArgumentException(
        "Invalid argument: Age should be a positive number.");
    }
    // Assign the new correct value
    this.age = value;
```

```
    }
}
```

In case someone tries to assign a value to **Age**, which is a negative number or 0, the code will throw an exception from the type **ArgumentException**, with details of the problem.

To protect itself from invalid data a class must **verify the input values for all properties and constructors** submitted to the setter methods, as well as all methods, which can change a field of a class. This programming practice to protect classes from invalid data and invalid internal states is widely used and is a part of the "Defensive Programming" concept, which we will discuss in chapter "High-Quality Programming Code".

Automatic Properties in C#

In C# we could define properties without explicitly defining the underlying field behind them. This is called **automatic properties**:

Point.cs

```
using System;

class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}

class PointTest
{
    static void Main()
    {
        Point myPoint = new Point(2, 3);
        Console.WriteLine("The X coordinate is: " + myPoint.X);
        Console.WriteLine("The Y coordinate is: " + myPoint.Y);
    }
}
```

The above example declares a class **Point** with two **automatic properties**: **X** and **Y**. These properties do not have explicitly defined underlying fields and the compiler defines them during the compilation. It looks like the **get** and **set** methods are empty but in fact the compiler defines an underlying field and fills the body of the **get** and **set** accessors with some code to read / write the automatically defined underlying field.

Use automatic properties for simple classes where you want to write less code but have in mind that when you use automatic properties your control over the assigned values is limited. You might have difficulties to add checks for invalid data.

Types of Properties

Depending on their definition we can classify the properties as follows:

1. **Read-only**, i.e. these properties have only a **get** method as shown by the area of the rectangle.
2. **Write-only**, i.e. these properties have only a **set** method, but no method for reading the value of the property.
3. And the most common case is **read-write**, where the property has **methods both for reading and for changing the value**.

Some properties are designed to be **read-only**. Others are supposed to support **both read and write operations**. The developers should decide whether someone should be able to change the value of given property and define it as read-only or read / write. **Write-only** properties are used very rarely.

Static Classes and Static Members

We call an element static when it is declared with the modifier **static**. In C# we can declare fields, methods, properties, constructors and classes as static.

We will first consider the **static elements** of a class or in other words we will look at the fields, methods, properties and constructors of a class and then we will study the concept of the static class.

What the Static Elements Are Used For?

Before we study the working principle of the static elements, let's see the reasons why we need to use them.

Method to Sum Two Numbers

Let's imagine that we have a class with a single method that always works in the same manner. For example, its task is to get two numbers and return their sum as a result. In this scenario there is no matter exactly which object of that class is going to implement that method since it will always do the same thing – adding two numbers together, independent of the calling object.

In practice **the behavior of the method does not depend of the object state** (the values in the object field). So why the need to create an object to accomplish that method provided that the method does not depend on any of the objects of that class? Why not just get the class to implement that method?

Instance Counter for Given Class

Consider a different scenario. Let's say we want to keep in our program the current number of objects, which have been created by a given class. How will we keep that variable, which stores **the number of created objects**?

As we know, we will not be able to store the variable as a class field because for each created object there will be created a copy of that field, initialized with default value. Every single object will store its own field for indication of the number of objects and the objects will not be able to **share information**.

It looks like the counter should be outside a class field rather than inside it. In the following subsections we will find out how to deal with such a problem.

What Is a Static Member?

Formally speaking, a **static member** of the class is every field, property, method or other member, which has a **static** modifier in its declaration[1]. That means that fields, methods and properties, marked as static, belong to the particular class rather than to any particular object of the given class.

Therefore, when we mark a field, method or property as **static**, we can use them without creating any object of the given class. All we need is to have access (visibility) to the class so that we can call the object's static methods or its static fields and properties.

	Static elements of the class can be used without creating an object of the given class.

On the other hand if we have created objects of the given class then its **static fields and properties will be shared** and there will be only one copy of the static field or property which will be shared among all objects of the given class. Because of that reason in the VB.NET language we have the keyword **shared** instead of the **static** keyword.

Static Fields

When we create objects from a given class, each of them holds different values in its fields. For example, consider again the class **Dog**:

```
public class Dog
{
   // Instance variables
   private string name;
```

```
    private int age;
}
```

There are two fields in the class, one for the name – **name** and another one for the age – **age**. Every object, each of these fields, has its own value, which is stored in a different place in the memory for every object.

Sometimes, however, we want to have **common fields** for all objects of a given class. To achieve that, we have to use the **static** modifier in the field declarations. As we already said, such fields are called **static fields**. In the literature they are also called **class variables**.

We say that the static fields are **class associated**, rather than associated with any object from the particular class. That means that all objects, created by the description of a class **share** the static fields of the class.

All objects, created by the description of a given class (that is, instances of a given class), share the static fields of the class.

Declaration of Static Fields

The static fields are declared the same way as the class fields. If there is access modifier, the keyword **static** should be added after it.

```
[<access_modifier>] static <field_type> <field_name>
```

Here is how a field named **dogCount** would look like. The field stores information about the count of the created objects from the class **Dog**:

Dog.cs

```
public class Dog
{
    // Static (class) variable
    static int dogCount;

    // Instance variables
    private string name;
    private int age;
}
```

The static fields are created when we try to access them for the first time (read / modify). After their creation they are initialized with their default values of their types.

Initialization during Declaration

If during the static field declaration we set an initialization value, it will be assigned to the particular static field. The **initialization executes only once** when the field is accessed for the first time right after the assignment has finished. The next time when the field is accessed that field initialization will not execute.

We can append the **static field initialization** in the example above:

```
// Static variable - declaration and initialization
static int dogCount = 0;
```

This initialization will complete during the first invocation to the static field. When we access some static field, an amount of memory will be reserved for it and it will be initialized with its default values. If the field has initialization during declaration (like it is in our case with the **dogCount** field) this initialization will execute. If we try later to access the field from other part of our program this process will not repeat, because the static field already exists and is initialized.

Accessing Static Fields

In contrast to the common (non-static) class fields, the static fields that are associated with the particular class can be accessed through an external class. In order to do that we need to put a **"dot" notation** this way:

```
<class_name>.<static_field_name>
```

For example, if we want to print the value of the static field that holds the number of created objects of our class **Dog** we should do that:

```
static void Main()
{
  // Access to the static variable through class name
  Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

The result of the **Main()** method is:

```
Dog count is now 0
```

If we have a method in the class, which is defined as a static field, we can access it directly without the class name, because it is known by default.

```
<static_field_name>
```

Modification of the Static Field Values

As we said before, the static variables are **shared between all objects** of the class and do not belong to any object of the particular class. That way any object can access and modify the static field values and in the same time other objects can "see" the modified values.

That's why if we want to count the number of created objects of the class **Dog**, we should use a **static field** and increment it by one every time the constructor is invoked, i.e. every time we create and object of our class.

```
public Dog(string name, int age)
{
  this.name = name;
  this.age = age;

  // Modifying the static counter in the constructor
  Dog.dogCount += 1;
}
```

We access static field from the class **Dog** so we can use the following code in order to access the field **dogCount**:

```
public Dog(string name, int age)
{
  this.name = name;
  this.age = age;

  // Modifying the static counter in the constructor
  dogCount += 1;
}
```

The first way is preferable, it is clear that the field in the class **Dog** is static. The code is more readable.

Let's create some objects of the class **Dog** and print out their number in order to check if we are right:

```
static void Main()
{
  Dog dog1 = new Dog("Jackie", 1);
  Dog dog2 = new Dog("Lassy", 2);
  Dog dog3 = new Dog("Rex", 3);

  // Access to the static variable
  Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

The output of the example is:

```
Dog count is now 3
```

Constants

Before we finish with the static fields we should get familiar with one more specific type of static fields.

Like the constants of mathematics, in C# special fields of a class called **constants** can be created. Once declared and initialized **constants always have the same value** for all objects of a particular type.

In C# constants are of two types:

1. Constants the values of which are extracted during the compilation of the program (**compile-time constants**).
2. Constants the values of which are extracted during the execution of the program (**run-time constants**).

Compile-Time Constants (const)

Constants, which are calculated at compile time (compile-time constants), are declared as follows, using modifier **const**:

```
[<access_modifiers>] const <type> <name>;
```

Constants, which are declared with special word **const**, are static fields. Nevertheless, the use of modifier **static** is not required (nor allowed by the compiler) in their declaration.

Although the constants declared with a modifier const are static fields, they must not and cannot use the static modifier in their declaration.

For example, if we want to declare as a constant the number "PI", which is known to us from mathematics, this can be done as follows:

```
public const double PI = 3.141592653589793;
```

The value we assign to a particular constant can be an expression, which has to be calculated by the compiler at compile time. For example, as we know from mathematics, the constant "PI" can be represented as the approximate result of the division of numbers 22 and 7:

```
public const double PI = 22d / 7;
```

When we try to print the value of the constant:

```
static void Main()
{
  Console.WriteLine("The value of PI is: " + PI);
}
```

The command line will display:

```
The value of PI is: 3.14285714285714
```

If we do not give a value to a constant at its declaration, but later, we will get a compilation error. For example, if we take the example of the constant PI, we first declare the constant and later try to give it a value:

```
public const double PI;

// … Some code …

public void SetPiValue()
{
  // Attempting to initialize the constant PI
  PI = 3.141592653589793;
}
```

The compiler will issue an error like this one, indicating the line, where the constant is declared:

```
A const field requires a value to be provided
```

Let’s pay attention, again:

Constants declared with modifier const must be initialized at the moment of their declaration.

Assigning Constant Values at Runtime

Having learned how to declare constants that are being initialized at compile time, let’s consider the following example: we want to create a class for color (**Color**). We will use the so-called **Red-Green-Blue (RGB) color model**, according to which, each color is represented by mixing the three primary colors – red, green and blue. These three primary colors are represented as three integers in the range from 0 to 255. For example, black is represented as (0, 0, 0), white as (255, 255, 255), blue – (0, 0, 255) etc.

In our class we declare three integer fields for red, green and blue light and a constructor that accepts values for each of them:

Color.cs

```
class Color
{
  private int red;
  private int green;
  private int blue;

  public Color(int red, int green, int blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }
}
```

As some colors are used more frequently than others (for example, black and white) we can **declare constants for them**, with the idea that the users of our class will take them for granted, instead of creating their own objects for these particular colors every time. To do this, we modify the code of our class as follows, adding the declaration of the following color-constants:

Color.cs

```
class Color
{
  public const Color Black = new Color(0, 0, 0);
  public const Color White = new Color(255, 255, 255);

  private int red;
  private int green;
  private int blue;

  public Color(int red, int green, int blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }
}
```

Strangely, when we try to compile, we **get the following error**:

```
'Color.Black' is of type 'Color'. A const field of a reference
type other than string can only be initialized with null.
```

```
'Color.White' is of type 'Color'. A const field of a reference
type other than string can only be initialized with null.
```

This is so because in C#, constants, declared with the modifier **const**, can be only of the following types:

1. Primitive types: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double**, **decimal**, **bool**.
2. **Enumerated types** (discussed in section "Enumerations" at the end of this chapter).
3. **Reference types** (mostly the type **string**).

The problem with the compilation of the class in our example is connected with the reference types and the restriction on the compiler not to allow simultaneous use of the operator **new** when declaring a constant when this constant is declared with the modifier **const**, unless the reference type can be calculated at compile time.

As we can guess, the only reference type, which can be calculated at compile time while using the operator **new** is **string**.

Therefore, the only possibilities for reference type constants that are declared with modifier **const** are, as follows:

1. The constants must be of type **string**.
2. The value, which we assign to the constant of reference type, other than **string**, is **null**.

We can formulate the following definition:

Constants declared with modifier const must be of primitive, enumeration or reference type, and if they are of reference type, this type must be either a string or the value, that we assign to the constant, must be null.

Thus, using the modifier **const**, we will not be able to declare the constants **Black** and **White** of type **Color** in our color class because they aren't **null**. The next section will show us how to deal with this problem.

Runtime Constants (readonly)

When we want to declare reference type constants, which cannot be calculated during compilation of the program, we must use a combination of **static readonly** modifiers, instead of **const** modifier.

```
[<access_modifiers>] static readonly <reference-type> <name>;
```

Accordingly, **<reference-type>** is a type the value of which cannot be calculated at compilation time.

The compilation is successful if we replace **const** by **static readonly** in the last example of the previous section:

```
public static readonly Color Black = new Color(0, 0, 0);
public static readonly Color White = new Color(255, 255, 255);
```

Naming the Constants

The constants names in C# follow the **PascalCase** rule according to the Microsoft's official C# coding convention. If the constant is composed of several words, each new word after the first one begins with a capital letter. Here are some examples of correctly named constants:

```
// The base of the natural logarithms (approximate value)
public const double E = 2.718281828459045;
public const double PI = 3.141592653589793;
public const char PathSeparator = '/';
public const string BigCoffee = "big coffee";
public const int MaxValue = 2147483647;
public static readonly Color DeepSkyBlue = new Color(0,104,139);
```

Sometimes naming in style **ALL-CAPS** is used but it is not officially supported by the Microsoft code conventions, even though it is widely distributed in programming:

```
public const double FONT_SIZE_IN_POINTS = 14; // 14pt font size
```

The examples made it clear that the difference between **const** and **static readonly** fields is in the moment of their value assignments. The compile-time constants (**const**) must be initialized at the moment of declaration, while the run-time constants (**static readonly**) can be initialized at a later stage, for example in one of the constructors of the class in which they are defined.

Using Constants

Constants are used in programming to **avoid repetition of numbers, strings or other common values** (literals) in the program and to enable them to change easily. The use of constants instead of brutally hardcoded repeating values facilitates readability and maintenance of the code and is highly recommended practice. According to some authors all literals other than **0**, **1**, **-1**, empty string, **true**, **false** and **null** must be declared as constants, but this can make it difficult to read and maintain the code instead of making it simple. Therefore, it is believed that **values, which occur more than once in the program or are likely to change over time, must be declared as constants**.

In the chapter "High-Quality Programming Code" will we learn in details when and how to use constants efficiently.

Static Methods

Like static fields, we declare a method as static if we want it to be associated only with the class and not with a particular class object.

Declaration of Static Methods

To declare a **static method** syntactically means that we must add the keyword **static** in the method's declaration:

```
[<access_modifier>] static <return_type> <method_name>()
```

Let's for example declare the method of summing two numbers, which we discussed at the beginning of this section:

```
public static int Add(int number1, int number2)
{
  return (number1 + number2);
}
```

Accessing Static Methods

Like static fields, static methods can be **accessed with the "dot" notation** (the dot operator) applied to the name of the class and the class name can be skipped if the calling is performed by the same class, in which the static method is declared. Here is an example of calling the static method **Add(…)**:

```
static void Main()
{
  // Call the static method through its class
  int sum = MyMathClass.Add(3, 5);

  Console.WriteLine(sum);
}
```

Access between Static and Non-Static Members

In most cases **static methods are used to access static fields** in the class they have been defined. For example, if we want to declare a method, which returns the number of the created objects of the **Dog** class, it must be static, because our counter will be static too:

```
public static int GetDogCount()
{
```

```
    return dogCount;
}
```

But when we examine how static and non-static methods and fields can be accessed, not all combinations are allowed.

Accessing Non-Static Members from Non-Static Method

Non-static methods can access non-static fields and other non-static methods of the class. For example, in the **Dog** class we can declare method **PrintInfo()**, which displays information about our dog:

Dog.cs

```
public class Dog
{
    // Static variable
    static int dogCount;

    // Instance variables
    private string name;
    private int age;

    public Dog(string name, int age)
    {
        this.name = name;
        this.age = age;

        dogCount += 1;
    }

    public void Bark()
    {
        Console.Write("wow-wow");
    }

    // Non-static (instance) method
    public void PrintInfo()
    {
        // Accessing instance variables – name and age
        Console.Write("Dog's name: " + this.name + "; age: "
            + this.age + "; often says: ");

        // Calling instance method
        this.Bark();
```

```
    }
}
```

Of course, if we create an object of the **Dog** class and call his **PrintInfo()** method:

```
static void Main()
{
    Dog dog = new Dog("Doggy", 1);
    dog.PrintInfo();
}
```

The result will be the following:

```
Dog's name: Doggy; age: 1; often says: wow-wow
```

Accessing Static Elements from Non-Static Method

We can access static fields and static methods of the class from non-static method. As we learned earlier, this is because static methods and variables are bound by class, rather than a specific method and the static elements can be accessed from any object of the class, even of external classes (as long as they are visible to them).

For example:

Circle.cs

```
public class Circle
{
    public static double PI = 3.141592653589793;

    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public static double CalculateSurface(double radius)
    {
        return (PI * radius * radius);
    }

    public void PrintSurface()
    {
```

```
        double surface = CalculateSurface(radius);
        Console.WriteLine("Circle's surface is: " + surface);
    }
}
```

In the example, we provide access to the value of the static field **PI** of the non-static method **PrintSurface()**, by calling the static method **CalculateSurface()**. Let's try to call this non-static method:

```
static void Main()
{
    Circle circle = new Circle(3);
    circle.PrintSurface();
}
```

After the compilation and the execution, the following result will be printed on the console:

```
Circle's surface is: 28.2743338823081
```

Accessing Static Elements of the Class from Static Method

We can call a static method or static field of the class from another static method without any problems.

For example, let's consider our class for mathematical calculations. We have declared the constant **PI**, in it. We can declare a static method for finding the length of the circle (the formula for finding perimeter of a circle is **2πr**, where **r** is the radius of the circle), that uses the constant **PI** for calculating the perimeter of a circle. Then, to show that static method can call another static method, we can call the static method for finding the perimeter of a circle from the static method **Main()**:

MyMathClass.cs

```
public class MyMathClass
{
    public const double PI = 3.141592653589793;

    // The method applies the formula: P = 2 * PI * r
    static double CalculateCirclePerimeter(double r)
    {
        // Accessing the static variable PI from static method
        return (2 * PI * r);
    }
```

```
    static void Main()
    {
        double radius = 5;

        // Accessing static method from other static method
        double circlePerimeter = CalculateCirclePerimeter(radius);

        Console.WriteLine("Circle with radius " + radius +
            " has perimeter: " + circlePerimeter);
    }
}
```

The code is compiled without errors and displays the following output:

```
Circle with radius 5.0 has perimeter: 31.4159265358979
```

Accessing Non-Static Elements from Static Method

Let's look at the most interesting case of a combination of accessing non-static and static elements of the class – **accessing non-static elements form a static method**.

We should know that from static method we can neither access non-static fields, nor call non-static methods. This is because static methods are bound to the class and do not "know" any object of the class. Therefore, the keyword **this** cannot be used in static methods – it is bound to a specific instance of the class. When we try to access non-static elements of the class (fields or methods) from static method, we will always get a compilation error.

Unauthorized Access to Non-Static Field – Example

If in our class **Dog** we try to declare a static method **PrintName()**, which returns as a result the value of the non-static field **name** declared in the class:

```
public static void PrintName()
{
    // Trying to access non-static variable from static method
    Console.WriteLine(name); // INVALID
}
```

Accordingly, the compiler will respond with an **error message**:

```
An object reference is required for the non-static field,
method, or property 'Dog.name'
```

If we try to access the field in the method, via the **keyword this**:

```
public void string PrintName()
{
  // Trying to access non-static variable from static method
  Console.WriteLine(this.name); // INVALID
}
```

The compiler will still not be satisfied and this time will **fail to compile** the class and will display the following message:

```
Keyword 'this' is not valid in a static property, static method,
or static field initializer
```

Illegal Call of Non-Static Method from Static Method – Example

Now we will try to call non-static method from static method. Let declare in our class **Dog**, the non-static method **PrintAge()**, which prints the value of the field **age**:

```
public void PrintAge()
{
  Console.WriteLine(this.age);
}
```

Accordingly, let's try from the method **Main()**, which we declare in the class **Dog**, to call this method without creating an object of our class:

```
static void Main()
{
  // Attempt to invoke non-static method from a static context
  PrintAge(); // INVALID
}
```

When we try to compile we will **get the following error**:

```
An object reference is required for the non-static field,
method, or property 'Dog.PrintAge()'
```

The result is similar, if we try to cheat the compiler, trying to call the method via the keyword **this**:

```
static void Main()
{
  // Attempt to invoke non-static method from a static context
  this.PrintAge(); // INVALID
}
```

Accordingly, as with the attempt to access the non-static field of a static method using the keyword **this**, the compiler displays the following error message and **fails to compile our class**:

```
Keyword 'this' is not valid in a static property, static method,
or static field initializer
```

From the examples, we can make the following conclusion:

Non-static elements of the class may NOT be used in a static context.

The problem with the access to non-static elements of the class of static method has a single solution – these non-static elements are accessed by reference to an object:

```
static void Main()
{
  Dog myDog = new Dog("Lassie", 2);
  string myDogName = myDog.name;
  Console.WriteLine("My dog \"" + myDogName +"\" has age of ");
  myDog.PrintAge();
  Console.WriteLine("years");
}
```

Accordingly, this code is compiled and the result is:

```
My dog "Lassie" has age of 2 years
```

Static Properties of the Class

Although rare, it is sometimes convenient to use and declare not the object characteristics, but the ones of the class. They possess the same characteristics like the properties related to the particular object of a particular class, which we discussed above, but with the difference that the **static properties refer to the class** (not its objects).

As we can guess, all we need to do to turn a simple property into a static one, is to **add the static keyword in its declaration**.

The static properties are **declared** as follows:

```
[<modifiers>] static <property_type> <property_name>
{
  // … Property's accessors methods go here
}
```

Let's consider an example. We have a class that describes a system. We can create many objects from it, but the model of the system has a version and a vendor, which are common to all instances created from this class. We can make the version and the vendors as static properties of the class:

SystemInfo.cs

```
public class SystemInfo
{
    private static double version = 0.1;
    private static string vendor = "Microsoft";

    // The "version" static property
    public static double Version
    {
        get { return version; }
        set { version = value; }
    }

    // The "vendor" static property
    public static string Vendor
    {
        get { return vendor; }
        set { vendor = value; }
    }

    // … More (non)static code here …
}
```

In this example we have chosen to keep the value of static properties in static variables (which are logical, since they are bound only to the class). The properties that we consider are **Version** and **Vendor**, respectively. For each of them we have created static methods for reading and modification. Thus, all objects of this class will be able to retrieve the current version and vendor of the system, which describes the class. Accordingly, if one day an upgrade of the system version is done and the value becomes **0.2**, as a result each object will receive the new version by accessing the class property.

Static Properties and the Keyword "this"

Like static methods, the keyword **this** cannot be used in the static properties, as the static property is associated only with the class and does not "recognize" objects of a class.

The keyword this cannot be used in static properties.

Accessing Static Properties

Like the static fields and methods, static properties can be accessed by **"dot" notation**, applied only to the name of the class in which they are declared.

To be sure, let's try to access the property **Version** through a variable of the class **SystemInfo**:

```
static void Main()
{
    SystemInfo sysInfoInstance = new SystemInfo();
    Console.WriteLine("System version: " +
        sysInfoInstance.Version);
}
```

When we try to compile the above code, we get the following error message:

```
Member 'SystemInfo.Version.get' cannot be accessed with an instance reference; qualify it with a type name instead
```

Accordingly, if we try to access the static properties through class name, the code compiles and works correctly:

```
static void Main()
{
    // Invocation of static property setter
    SystemInfo.Vendor = "Microsoft Corporation";

    // Invocation of static property getters
    Console.WriteLine("System version: " + SystemInfo.Version);
    Console.WriteLine("System vendor: " + SystemInfo.Vendor);
}
```

The code is compiled and the result of its execution is:

```
System version: 0.1
System vendor: Microsoft Corporation
```

Before proceeding to the next section, let's look at the printed value of the property **Vendor**. It is "**Microsoft Corporation**", although we have initialized it with the value "**Microsoft**" in the **SystemInfo** class. This is because we changed the value of the property **Vendor** of the first line of the **Main()** method, by calling its method of modification.

Static properties can be accessed only through dot notation, applied to the name of the class in which they are declared.

Static Classes

For complete understanding we have to explain that we can also declare classes as static. Similar to static members, a class is static, when the keyword **static** is used in its declaration.

```
[<modifiers>] static class <class_name>
{
  // … Class body goes here
}
```

When a class is declared as static, it is an indication that **this class contains only static members** (i.e. static fields, methods, properties) and cannot be instantiated.

The use of static classes is rare and most often associated with the **use of static methods and constants**, which do not belong to any particular object. For this reason, the details of static classes go beyond the scope of this book. Curious reader can find more information on the site of the Microsoft Developer Network (MSDN): http://msdn.microsoft.com/en-us/library/79b3xss3.aspx.

Static Constructors

To finish the section on static class members, we should mention that classes may also have **static constructor** (i.e. constructor that has the **static** keyword in its declaration):

```
[<modifiers>] static <class_name>([<parameters_list>])
{
}
```

Static constructors can be declared both in static and in non-static classes. They are **executed only once** when the first of the following two events occurs for the first time:

1. An object of class is created.
2. A static element of the class is accessed (field, method, property).

Most often static constructors are used for initialization of static fields.

Static Constructor – Example

Consider an example for the **use of a static constructor**. We want to make a class that quickly calculates the square root of an integer and returns the whole part of the result, which is also an integer. Since calculating the square root is a time-consuming mathematical operation involving calculations with real numbers and calculating convergent series, it is a good idea these calculations to be done once at program startup and then to use the already

calculated values. Of course, to make such **pre-computing of all square roots** in a given range, we must first define this range and it should not be too wide (e.g. from 1 to 1000). Then we need, at first request for a square roots of a number, to recalculate all the square roots in this range and then to return the already calculated value. Upon a following request for a square root, all values in this range will have already been calculated and returned directly. If the program is never required to calculate the square root, preliminary calculations should not be fulfilled at all.

Through the described process initially some CPU time is invested for preliminary calculations, but then the extraction of the square root later is done very quickly. If we have multiple calculations of the square root, the pre-calculation will significantly increase the performance.

All this can be implemented in one **static class with a static constructor**, in which the square roots will be recalculated. The results, which have already been calculated, can be **stored in a static array**. A **static method** can be used to extract the already pre-calculated value. Since the preliminary calculations are being performed in the static constructor, if the class for pre-calculated square roots is not used, they will not be executed and CPU time and memory will be saved.

This is how the implementation might look like:

```
static class SqrtPrecalculated
{
  public const int MaxValue = 1000;

  // Static field
  private static int[] sqrtValues;

  // Static constructor
  static SqrtPrecalculated()
  {
    sqrtValues = new int[MaxValue + 1];
    for (int i = 0; i < sqrtValues.Length; i++)
    {
      sqrtValues[i] = (int)Math.Sqrt(i);
    }
  }

  // Static method
  public static int GetSqrt(int value)
  {
    if ((value < 0) || (value > MaxValue))
    {
      throw new ArgumentOutOfRangeException(String.Format(
```

```
            "The argument should be in range [0...{0}].",
            MaxValue));
        }
        return sqrtValues[value];
    }
}

class SqrtTest
{
    static void Main()
    {
        Console.WriteLine(SqrtPrecalculated.GetSqrt(254));
        // Result: 15
    }
}
```

Structures

In C# and .NET Framework there are two implementations of the concept of "class" from the object-oriented programming: **classes** and **structures**. Classes are defined through the keyword **class** while the structures are defined through the keyword **struct**. The main difference between a structure and a class is that:

- **Classes are reference types** (references to some address in the heap which holds their members).
- **Structures (structs) are value types** (they directly hold their members in the program execution stack).

Structure (struct) – Example

Let's define a **structure** to hold a point in the 2D space, similar to the class **Point** defined in the section "Example of Encapsulation":

Point2D.cs

```
struct Point2D
{
    private double x;
    private double y;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
```

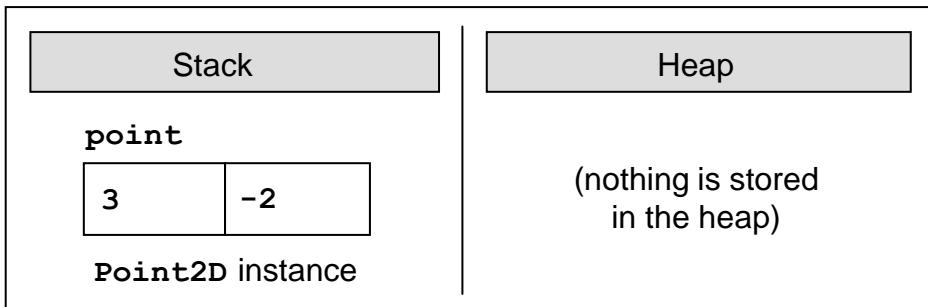
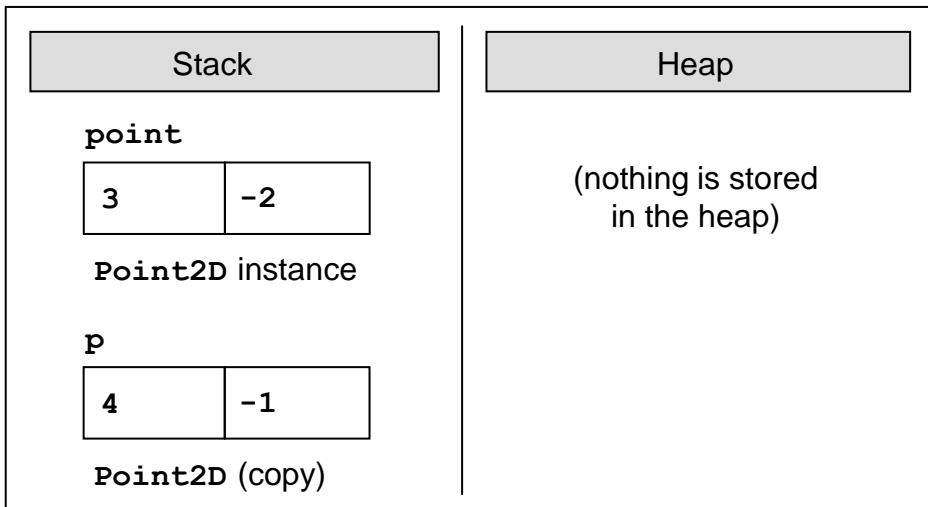
```
    }

    public double X
    {
        get { return this.x; }
        set { this.x = value; }
    }

    public double Y
    {
        get { return this.y; }
        set { this.y = value; }
    }
}
```

The only difference is that now we defined **Point2D** as **struct**, not as **class**. **Point2D** is a structure, a value type, so its instances behave like **int** and **double**. They are value types (not objects), which means they cannot be **null** and they are **passed by value** when taken as a method parameters.

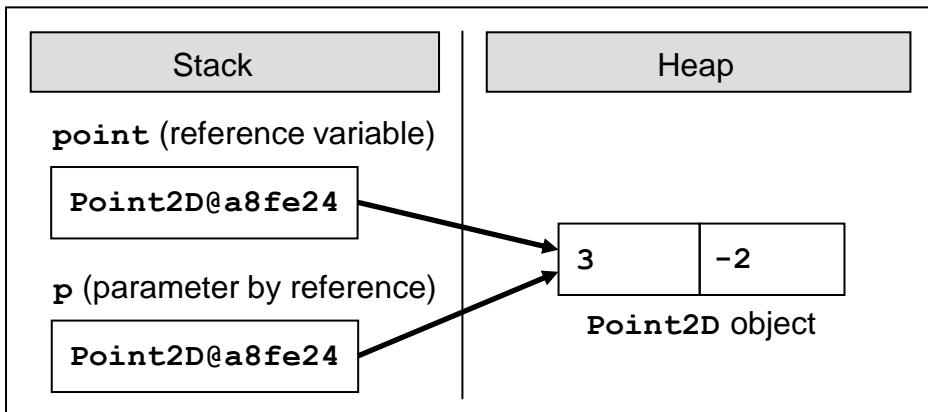
Structures are Value Types

Unlike classes, the **structures are value types**. To illustrate this we will play a bit with the **Point2D** structure:

```
class PlayWithPoints
{
    static void PrintPoint(Point2D p)
    {
        Console.WriteLine("({0},{1})", p.X, p.Y);
    }

    static void TryToChangePoint(Point2D p)
    {
        p.X++;
        p.Y++;
    }

    static void Main()
    {
        Point2D point = new Point2D(3, -2);
        PrintPoint(point);
        TryToChangePoint(point);
        PrintPoint(point);
    }
```

```
}
```

If we run the above example, the result will be as follows:

```
(3,-2)
(3,-2)
```

Obviously the **structures are value types** and when passed as parameters to a method **their fields are copied** (just like **int** parameters) and when changed inside the method, the change affects only the copy, not the original. This can be illustrated by the next few figures.

First, the **point** variable is created which holds a value of (3, -2):

Next, the method **TryToChangePoint(Point2D p)** is called and it copies the value of the variable **point** into **another place in the stack**, allocated for the parameter **p** of the method. When the parameter **p** is changed in the method's body, it is modified in the stack and this **does not affect the original variable point** which was previously passed as argument when calling the method:

If we change **Point2D** from **struct** to **class**, the result will be very different:

```
(3,-2)
(4,-1)
```

This is because the variable **point** will be now passed by reference (not by value) and its value will be shared between **point** and **p** in the heap. The figure below illustrates what happens in the memory at the end of the method **TryToChangePoint(Point2D p)** when **Point2D** is a class:

Class or Structure?

How to decide **when to use a class and when a structure**? We will give you some general guidelines.

Use structures to hold simple data structures consisting of few fields that come together. Examples are coordinates, sizes, locations, colors, etc. Structures are not supposed to have functionality inside (no methods except simple ones like **ToString()** and comparators). Use structures for **small data structures consisting of set of fields** that should be passed by value.

Use classes for more complex scenarios where you combine data and programming logic into a class. If you have logic, use a class. If you have more than few simple fields, use a class. If you need to pass variables by reference, use a class. If you need to assign a **null** value, prefer using a class. If you prefer working with a reference type, use a class.

Classes are used more often than structures. Use structs as exception, and **only if you know well what are you doing**!

There are few other **differences between class and structure** in addition that classes are reference types and structures are values types, but we will not going to discuss them. For more details refer to the following article in MSDN: http://msdn.microsoft.com/en-us/library/ms173109.aspx.

Enumerations

Earlier in this chapter we discussed what constants are, how to declare and use them. In this connection we will now consider a part of the C# language, in which a variety of logically connected constants can be linked by means of language. These language constructs are the so-called **enumerated types**.

Declaration of Enumerations

Enumeration is a structure, which resembles a class but differs from it in that in the class body we can **declare only constants**. Enumerations can take values only from the constants listed in the type. An enumerated variable can have as a value one of the listed in the type constants but cannot have value **null**.

Formally speaking, the enumerations can be declared using the reserved word **enum** instead of **class**:

```
[<modifiers>] enum <enum_name>
{
  constant1 [, constant2 [, [, … [, constantN]]
}
```

Under **<modifiers>** we understand the access modifiers **public**, **internal** and **private**. The identifier **<enum_name>** follows the rules for class names in C#. Constants separated by commas are declared in the enumeration block.

Consider an example. Let's define an enumeration for the days of the week (we will call it **Days**). As we can guess, the constants that will appear in this particular enumeration are the **names of the week days**:

Days.cs

```
enum Days
{
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
}
```

Naming of constants in one particular enumeration follows the same principles of naming of which we already explained in the "Naming Constants" section.

Note that each of the constants listed in the enumeration is of type this enumeration, i.e. in our case **Mon** belongs to type **Days**, as well as each of the other constants.

In other words, if we execute the following line:

```
Console.WriteLine(Days.Mon is Days);
```

This will be printed as a result:

```
True
```

Let's repeat again:

The enumerations are a set of constants of type – this listed type.

Nature of Enumerations

Each constant, which is declared in one enumeration, is being associated with a certain integer. By default, for this hidden integer representation of constants in one enumeration **int** is being used.

To show **"the integer nature" of constants** in the listed types let's try to figure out what's the numerical representation of the constant, which corresponds to "Monday" from the example of the previous subsection:

```
int mondayValue = (int)Days.Mon;
Console.WriteLine(mondayValue);
```

After we execute it, the result will be:

```
0
```

The values, associated with constants of a particular enumerated type by default are the indices in the list of constants of this type, i.e. numbers from 0 to the number of constants in the type less 1. In this way, if we consider the example with the enumeration type for the week days, used in the previous subsection, the constant **Mon** is associated with the numerical value 0, the constant **Tue** with the integer value 1, **Wed** – with 2, etc.

Each constant in one enumeration is actually a textual representation of an integer. By default this number is the constant's index in the list of constants of a particular enumeration type.

Despite the integer nature of constants in a particular enumeration, when we try to print a particular constant, its textual representation at the time of the constant's declaration will be printed:

```
Console.WriteLine(Days.Mon);
```

After we execute the code above we will get the following result:

```
Mon
```

Hidden Numerical Value of Constants in Enumeration

As we can guess it is possible to change the **numerical value of constants in an enumeration**. This is done when we assign the values we prefer to each of the constants at the time of declaration.

```
[<modifiers>] enum <enum_name>
{
  constant1[=value1] [, constant2[=value2] [, … ]]
}
```

Accordingly **value1**, **value2**, etc. must be integers.

To get a clearer idea of the given definition consider the following example: let's have a class **Coffee**, which represents a cup of coffee that customers order in a coffee shop:

Coffee.cs

```
public class Coffee
{
  public Coffee()
  {
  }
}
```

In this facility customers can order different amounts of coffee, as the coffee machine has predefined values "small" – 100 ml, "normal" – 150 ml and "double" – 300 ml. Therefore, we can declare one enumeration **CoffeeSize**, which has respectively three constants – **Small**, **Normal** and **Double**, the correspondent qualities of which will be assigned:

CoffeeSize.cs

```
public enum CoffeeSize
{
  Small=100, Normal=150, Double=300
}
```

Now we can add a field and property to the class **Coffee**, which reflect the type of coffee the customer has ordered:

Coffee.cs

```
public class Coffee
{
  public CoffeeSize size;
```

```
    public Coffee(CoffeeSize size)
    {
        this.size = size;
    }

    public CoffeeSize Size
    {
        get { return size; }
    }
}
```

Let's try to print the values of the coffee quantity for a normal and for one double coffee:

```
static void Main()
{
    Coffee normalCoffee = new Coffee(CoffeeSize.Normal);
    Coffee doubleCoffee = new Coffee(CoffeeSize.Double);

    Console.WriteLine("The {0} coffee is {1} ml.",
        normalCoffee.Size, (int)normalCoffee.Size);
    Console.WriteLine("The {0} coffee is {1} ml.",
        doubleCoffee.Size, (int)doubleCoffee.Size);
}
```

As we compile and execute this method, the following will be printed:

```
The Normal coffee is 150 ml.
The Double coffee is 300 ml.
```

Use of Enumerations

The main purpose of the enumerations is to **replace the numeric values**, which we would use, if there were no enumeration types. In this way the code becomes simpler and easier to read.

Another very important application of the enumerations is the pressure exercised by the compiler to use constants from the enumerations and not just numbers. Thus we minimize future errors in the code. For example, if we use an **int** variable instead of a variable from enumerations and a set of constants for the valid values, nothing prevents us from assigning the variable any value, e.g. -6723.

To make this clearer, consider the following example: create a class "**coffee price calculator**", which is calculating the price of each type of coffee, offered in the coffee shop:

PriceCalculator.cs

```
public class PriceCalculator
{
  public const int SmallCoffeeQuantity = 100;
  public const int NormalCoffeeQuantity = 150;
  public const int DoubleCoffeeQuantity = 300;

  public CashMachine() { }

  public double CalcPrice(int quantity)
  {
    switch (quantity)
    {
      case SmallCoffeeQuantity:
        return 0.20;
      case NormalCoffeeQuantity:
        return 0.30;
      case DoubleCoffeeQuantity:
        return 0.60;
      default:
        throw new InvalidOperationException(
          "Unsupported coffee quantity: " + quantity);
    }
  }
}
```

We have three constants for the capacity of the coffee cups in the coffee shop, respectively 100, 150 and 300 ml. Furthermore, **we expect** that users of our class will diligently use the defined constants, instead of numbers – **SmallCoffeeQuantity**, **NormalCoffeeQuantity** and **DoubleCoffeeQuantity**. The method **CalcPrice(int)** returns the respective price, calculating it by the submitted amount.

The problem lies in the fact that someone may decide not to use the constants defined by us and may submit an invalid number as a parameter of our method, for example: -1 or 101. In this case, if the method does not check for invalid quantity, it will likely return a wrong price, which is incorrect behavior.

To avoid this problem we will use one feature of these enumerations, namely constants in the enumeration type can be used in **switch-case** structures. They can be submitted as values of the operator **switch** and accordingly – as operands of the operator **case**.

The constants of enumerations can be used in switch-case structures.

Let's rework the method, which calculates the price for a cup of coffee, depending on the capacity of the cup. This time we will use the enumeration type **CoffeeSize**, which we declared in previous examples:

```
public double CalcPrice(CoffeeSize coffeeSize)
{
   switch (coffeeSize)
   {
      case CoffeeSize.Small:
         return 0.20;
      case CoffeeSize.Normal:
         return 0.40;
      case CoffeeSize.Double:
         return 0.60;
      default:
         throw new InvalidOperationException(
            "Unsupported coffee quantity: " + (int)coffeeSize);
   }
}
```

As we can see in this example, the possibility for the users of our method to provoke unexpected behavior of the method is negligible, because we force them to use specific values which to be used as arguments, namely constants of enumerated **CoffeeSize** type. This is one of the advantages of constants, which are declared in enumeration types to constants declared in any class.

Whenever possible, use enumerations instead of set of constants declared in a class.

Before we finish with the enumeration section we should mention that the enumerations are to be used with caution when working with the **switch-case** construct. For example, if one day the owner of the coffee shop buys many big cups (mugs) for coffee, we will need to add a new constant in the constant list of the enumeration **CoffeeSize**, which may be called, for example, **Overwhelming**:

CoffeeSize.cs

```
public enum CoffeeSize
{
   Small=100, Normal=150, Double=300, Overwhelming=600
}
```

When we try to calculate the coffee price with the new quantity, the method, which calculates the price, will throw an exception, informing the user that such amount of coffee is not available in the coffee shop.

What we should do to solve this problem is to add a new **case**-condition, which reflects the new constant in the enumerated **CoffeeSize** type.

	When we modify the list of constants in an existing enumeration, we should be careful not to break the logic of the code that already exists and uses the constants, declared so far.

Inner Classes (Nested Classes)

In C# an inner (nested) class is called a **class that is declared inside the body of another class**. Accordingly, the class that encloses the inner class is called an **outer class**.

The main reason to declare one class into another are:

1. To **better organize the code** when working with objects in the real world, among which have a special relationship and one cannot exist without the other.
2. To **hide a class in another class**, so that the inner class cannot be used outside the class wrapped it.

In general, inner classes are used rarely, because they complicate the structure of the code and increase the nested levels.

Declaration of Inner Classes

The inner classes are declared in the same way as normal classes, but are **located within another class**. Allowed modifiers in the declaration of the class are:

1. **public** – an inner class is accessible from any assembly.
2. **internal** – an inner class is available in the current assembly, in which is located the outer class.
3. **private** – access is restricted only to the class holding the inner class.
4. **static** – an inner class contains only static members.

There are four more permitted modifiers – **abstract, protected, protected internal**, **sealed** and **unsafe**, which are outside the scope and subject of this chapter and will not be considered here.

The keyword **this** to an inner class has relation only to the internal class, but not to the outside. Fields of the outside class **cannot be accessed** using the reference **this**. If necessary fields of the outer class can be accessed by the

internal, it needs in creating the internal class to submit a reference to an outer class.

Static members (fields, methods, properties) of the outer class **are accessible from the inner class** regardless of their level of access.

Inner Classes – Example

Consider the following example:

OuterClass.cs

```
public class OuterClass
{
  private string name;

  private OuterClass(string name)
  {
    this.name = name;
  }

  private class NestedClass
  {
    private string name;
    private OuterClass parent;

    public NestedClass(OuterClass parent, string name)
    {
      this.parent = parent;
      this.name = name;
    }

    public void PrintNames()
    {
      Console.WriteLine("Nested name: " + this.name);
      Console.WriteLine("Outer name: " + this.parent.name);
    }
  }

  static void Main()
  {
    OuterClass outerClass = new OuterClass("outer");
    NestedClass nestedClass = new
      OuterClass.NestedClass(outerClass, "nested");
    nestedClass.PrintNames();
  }
}
```

In the example the outer class **OuterClass** defines into itself as a member the class **InnerClass**. Non-static inner class methods have access to their own body **this** as well as the instance of outside class **parent** (through syntax **this.parent**, if the **parent** reference is added by the developer). In the example while creating the inner class, **parent** reference is set to constructor of the outer class.

If we run the above example, we will obtain the following result:

```
Nested name: nested
Outer name: outer
```

Usage of Inner Classes

Consider an example. Let's have a class for car – **Car.** Each car has an engine and doors. Unlike the car's door, however, the engine makes no sense regarded as being outside the car, because without it, the car cannot run, i.e. we have composition (see the section "Class Diagrams: Composition" in the chapter "Principles of Object-Oriented Programming").

	When the connection between the two classes is a composition, the class, which consequently is a part of another class, is convenient to be declared as inner class.

Therefore, if you declare the class for a car: **Car** would be appropriate to create an inner class **Engine,** which will reflect the appropriate concept for the car engine:

Car.cs

```
class Car
{
   Door FrontRightDoor;
   Door FrontLeftDoor;
   Door RearRightDoor;
   Door RearLeftDoor;
   Engine engine;

   public Car()
   {
      engine = new Engine();
      engine.horsePower = 2000;
   }

   public class Engine
   {
```

```
        public int horsePower;
    }
}
```

Declare Enumeration in a Class

Before proceeding to the next section that refers to generic types, it should be noticed, that sometimes **enumeration should and can be declared within a class** in order of better encapsulation of the class.

For example, the enumeration of type **CoffeeSize**, we have created in the previous section, can be declared inside the body of the class **Coffee**, thereby it improves its encapsulation:

Coffee.cs

```
class Coffee
{
    // Enumeration declared inside a class
    public static enum CoffeeSize
    {
        Small = 100, Normal = 150, Double = 300
    }

    // Instance variable of enumerated type
    private CoffeeSize size;

    public Coffee(CoffeeSize size)
    {
        this.size = size;
    }

    public CoffeeSize Size
    {
        get { return size; }
    }
}
```

Respectively, the method for calculation of the price of coffee will be slightly modified slightly:

```
public double CalcPrice(Coffee.CoffeeSize coffeeSize)
{
    switch (coffeeSize)
    {
```

```
      case Coffee.CoffeeSize.Small:
        return 0.20;
      case Coffee.CoffeeSize.Normal:
        return 0.40;
      case Coffee.CoffeeSize.Double:
        return 0.60;
      default:
        throw new InvalidOperationException(
          "Unsupported coffee quantity: " + ((int)coffeeSize));
    }
}
```

Generics

In this section we will explain the concept of **generic classes** (generic data types, generics). Before we begin, however, let's look through an example that will help us understand more easily the idea.

Shelter for Homeless Animals – Example

Let's assume that we have two classes. A class **Dog**, which describes a dog:

Dog.cs

```
public class Dog
{
}
```

And let a class **Cat**, which describes a cat:

Cat.cs

```
public class Cat
{
}
```

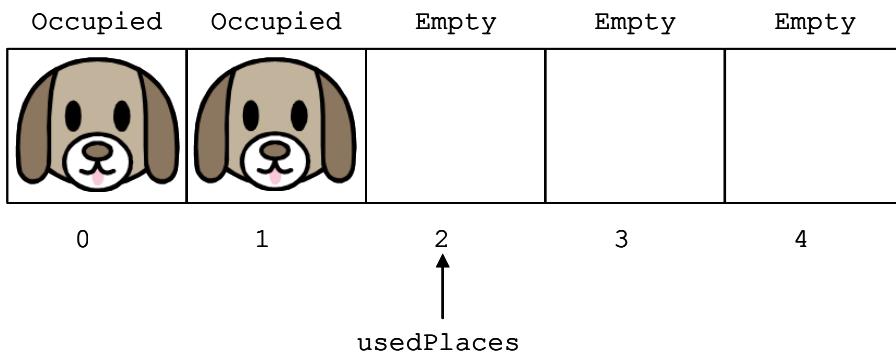
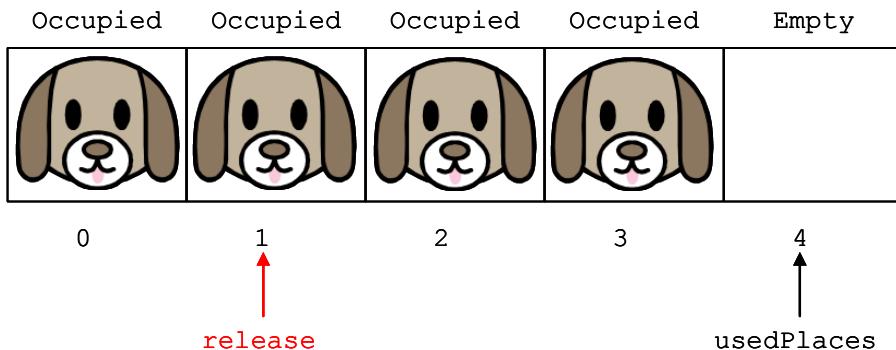
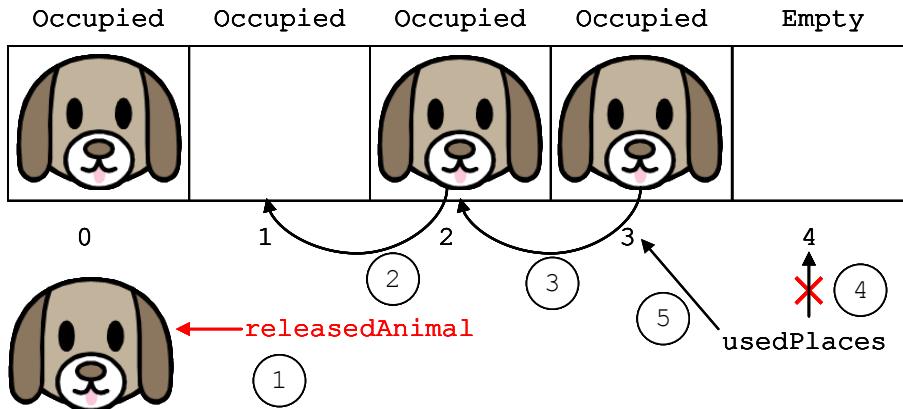
Then we want to create a class that describes a **shelter for homeless animals – AnimalShelter**. This class has a specific number of free cells, which determines the number of animals, which could find refuge in the shelter. The special feature of the class, that we want to create, is that it only needs to accommodate animals of the same kind, in our case, dogs or cats only, because the coexistence of different species is not always a good idea.

If we think about how to solve the task with the knowledge that we have until here, we will come to the following conclusion – to ensure that our class will contain elements only from one and the same type we need to use an array of

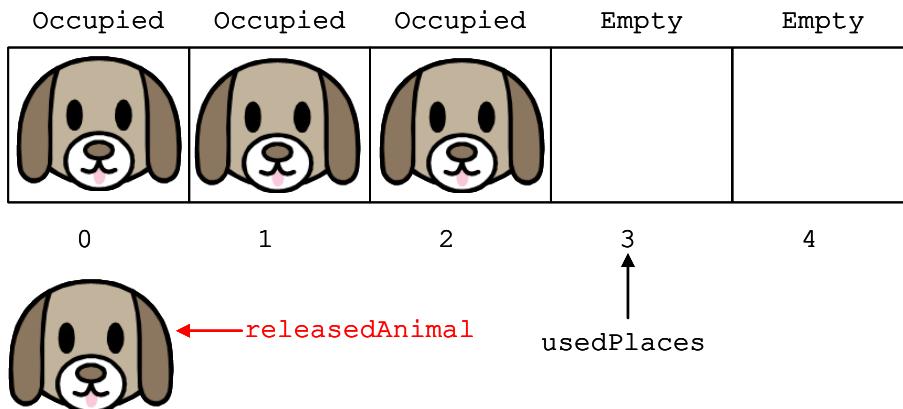
identical objects. These objects may be dogs, cats or simply instances of the universal type **object**.

For instance, if we want to make a shelter for dogs, here is how our class would look like:

AnimalsShelter.cs

```
public class AnimalShelter
{
   private const int DefaultPlacesCount = 20;

   private Dog[] animalList;
   private int usedPlaces;

   public AnimalShelter() : this(DefaultPlacesCount)
   {
   }

   public AnimalShelter(int placesCount)
   {
      this.animalList = new Dog[placesCount];
      this.usedPlaces = 0;
   }

   public void Shelter(Dog newAnimal)
   {
      if (this.usedPlaces >= this.animalList.Length)
      {
         throw new InvalidOperationException("Shelter is full.");
      }
      this.animalList[this.usedPlaces] = newAnimal;
      this.usedPlaces++;
   }

   public Dog Release(int index)
   {
      if (index < 0 || index >= this.usedPlaces)
      {
         throw new ArgumentOutOfRangeException(
            "Invalid cell index: " + index);
      }
      Dog releasedAnimal = this.animalList[index];
      for (int i = index; i < this.usedPlaces - 1; i++)
      {
         this.animalList[i] = this.animalList[i + 1];
```

```
		}
		this.animalList[this.usedPlaces - 1] = null;
		this.usedPlaces--;

		return releasedAnimal;
	}
}
```

Shelter capacity (number of animals, which it is capable to accommodate) is set when the object is created. By default it is the value of the constant **DefaultPlacesCount**. We use the field **usedPlaces** to monitor the occupied cells (at the same time we use it to index into the array for "pointing" to the first space from left to right in the array).

We have created a method for adding a new dog into the shelter – **Shelter()** and respectively for releasing from the shelter – **Release(int)**.

The method **Shelter()** adds each new animal in the first free cell in the right side of the array (if there is any free).

The method **Release(int)** accepts the number of cell from which the dog will be released (i.e. the index number in the array, where it is stored a link to the object of type **Dog**).

Then it moves all animals which are having a bigger cell number then the current cell, from which we will release a dog, with a position to the left (steps 2 and 3 are shown in the diagram below).

Released cell at position **usedPlaces-1** is marked as free, and a value **null** is assigned to it. This provides release of the reference to it and respectively allows the system to clean memory (garbage collector), to release the object if it is not used anywhere else in the program at this moment. This prevents from indirect loss of memory (memory leak).

Finally, it assigns the number of the last free cell to a **usedPlaces** field (steps 4 and 5 of the scheme above).

It is visible that the "removal" of an animal from a cell **could be a slow operation**, because it requires the transfer of all animals from the next cells with one position left. In the chapter "Linear Data Structures" we will discuss also more efficient ways of presenting the animal shelter, but for now let's focus on the topic about generic types.

So far we succeed implementing functionality of the shelter – the class **AnimalShelter**. When we work with objects of type **Dog**, everything compiles and executes smoothly:

```
static void Main()
{
  AnimalShelter dogsShelter = new AnimalShelter(10);
  Dog dog1 = new Dog();
  Dog dog2 = new Dog();
  Dog dog3 = new Dog();

  dogsShelter.Shelter(dog1);
  dogsShelter.Shelter(dog2);
  dogsShelter.Shelter(dog3);

  dogsShelter.Release(1); // Releasing dog2
}
```

What happens, however, if we attempt to use an **AnimalShelter** class for objects of type **Cat**:

```
static void Main()
{
  AnimalShelter dogsShelter = new AnimalShelter(10);

  Cat cat1 = new Cat();

  dogsShelter.Shelter(cat1);
}
```

As expected, the **compiler displays an error**:

```
The best overloaded method match for 'AnimalShelter.Shelter(
Dog)' has some invalid arguments. Argument 1: cannot convert
from 'Cat' to 'Dog'
```

Consequently, if we want to create a shelter for cats, we will not be able to reuse the class that we already created, although the operations of adding and removing animals from the shelter will be identical. Therefore, we have to literally copy **AnimalShelter** class and change only the type of the objects, which are handled – **Cat**.

Yes, but if we decide to make a shelter for other species? How many classes of shelters for the particular type of animals we shall create?

We can see that this solution of the problem **is not sufficiently comprehensive** and does not fully meets the terms, which we were set – to exist a **single class** that describes our shelter **for any kind of animal** (i.e. for all objects) and by working with it, **it should contain only one kind of animals** (i.e. only objects of one and the same type).

We could use instead of the type **Dog**, the universal type **object**, which can take values as **Dog**, **Cat** and all other data types, but this will create some inconvenience, associated with the need to convert back from the **object** to the **Dog**, when creating a shelter for dogs and it contains cells of type **object**, instead of type **Dog**.

To solve the task efficiently, we have to use a feature of the C# language that allows us to satisfy all required conditions simultaneously. It is called **generics** (template classes).

What Is a Generic Class?

As we know if a method needs additional information to operate properly, this information is passed to the method using parameters. During the execution of the program, when calling this particular method, we pass arguments to the method, which are assigned to its parameters and then used in the method's body.

Like the methods, when we know, that the functionality (actions) encapsulated into a class, can be applied not only to objects of one, but to many (heterogeneous) types, and these types are not known at the time of declaring the class, we can use a functionality of the language C# called **generics** (generic types).

It allows us to **declare parameters of this class, by indicating an unknown type** that the class will work eventually with. Then, when we instantiate our generic class, we replace the unknown with a particular. Accordingly, the newly created object will only work with objects of this type that we have assigned at its initialization. The specific type can be any data type that the compiler recognizes, including class, structure, enumeration or another generic class.

To get a cleaner picture of the nature of the generic types, let's return to our task from the previous section. As you might guess, the class that describes the animal shelter (**AnimalShelter**) **can operate with different types of animals**. Consequently, if we want to create a general solution of the task, during the declaration of class **AnimalShelter**, we cannot know what type of animals will be sheltered to shelter. This is sufficient indication, that we can typify our class, adding to the declaration of the class as a parameter, the unknown type of animals.

Later, when we want to create a dog's shelter for example, this parameter of the class will pass the name of our type – class **Dog**. Accordingly, if you create a shelter for cats, we will pass the type **Cat**, etc.

	Typifying a class (creating a generic class) means to add to the declaration of a class a parameter (replacement) of unknown type, which the class will use during its operation. Subsequently, when the class is instantiated, this parameter is replaced with the name of some specific type.

In the following sections we will introduce the syntax of generic classes and we will modify our previous example to use generics.

Declaration of Generic Class

Formally, the **parameterizing of a class** is done by adding **<T>** to the declaration of the class, after its name, where **T** is the substitute (parameter) of the type, which will be used later:

```
[<modifiers>] class <class_name><T>
{
}
```

It should be noticed that the characters '**<**' and '**>**', which surround the substitution **T** are an obligatory part of the syntax of language C# and must participate in the declaration of a generic class.

The **declaration of generic class**, which describes a shelter for homeless animals, should look like as follows:

```
class AnimalShelter<T>
{
  // Class body here …
}
```

Let's can imagine that we are creating a **template of our class AnimalShelter**, which we will specify later, replacing **T** with a specific type, for instance a **Dog**.

A particular class may have more than one substitute (to be parameterized by more than one type), depending on its needs:

```
[<modifiers>] class <class_name><T1 [, T2, [… [, Tn]]]>
{
}
```

If the class needs **several different unknown types**, these types should be listed by a comma between the characters '**<**' and '**>**' in the declaration of the class, as each of the substitutes used must be different identifier (e.g. a different letter) – in the definition they are indicated as **T1**, **T2**, …, **Tn**.

In case, we should to create a shelter for animals of a mixed type, one that accommodates both – dogs and cats, we should declare the class as follows:

```
class AnimalShelter<T, U>
{
  // Class body here …
}
```

If this were our case, we would use the first parameter **T**, to indicate objects of type **Dog**, which our class would operate with, and with **U** – to indicate objects of type **Cat**.

Specifying Generic Classes

Before we present more details about generics, we should look at **how to use generic classes.** The using of generic classes should be done as follows:

```
<class_name><concrete_type><variable_name> =
  new <class_name><concrete_type>();
```

Again, similar to **T** substitution in the declaration of our class, the characters '**<**' and '**>**' surrounding a particular class **concrete_type**, are required.

If we want to create two shelters, one for dogs and one for cats, we should use the following code:

```
AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>();
AnimalShelter<Cat> catsShelter = new AnimalShelter<Cat>();
```

In this way, we ensure that the shelter **dogsShelter** will always contain objects of a type **Dog** and the variable **catsShelter** will always operate with objects of type **Cat.**

Using Unknown Types by Declaring Fields

Once used during the class declaration, the parameters that are used to indicate the unknown types are visible in the whole body of the class, therefore they can be used to declare the field as each other type:

```
[<modifiers>] T <field_name>;
```

As we can guess, in our example with shelter for homeless animals, we can use this feature provided by language C#, to declare the type of field **animalsList**, which holds references to objects for the housed animals, instead of a specific type of **Dog**, with parameter **T**:

```
private T[] animalList;
```

Let's assume when we create an object of our class, setting a specific type (e.g. **Dog**) during the execution of the program, t**he unknown type T will be replaced** with the above type. If we choose to create a shelter for dogs, we can consider that our field is declared as follows:

```
private Dog[] animalList;
```

Accordingly, when we want to initialize a particular field in the constructor of our class, we should do it as usual – creating an array, using substitution of the unknown type – **T**:

```
public AnimalShelter(int placesNumber)
{
  animalList = new T[placesNumber]; // Initialization
  usedPlaces = 0;
}
```

Using Unknown Types in a Method's Declaration

As an **unknown type** used in the declaration of a generic class is visible from opening to closing brace of the class body, except for field's declaration, it **can be used in a method declaration**, namely:

As a parameter in the list of parameters of the method:

```
<return_type> MethodWithParamsOfT(T param)
```

- As a result of implementation of the method:

```
T MethodWithReturnTypeOfT(<params>)
```

As we already guessed, using our example, we can adapt the methods **Shelter(…)** and **Release(…)**, respectively:

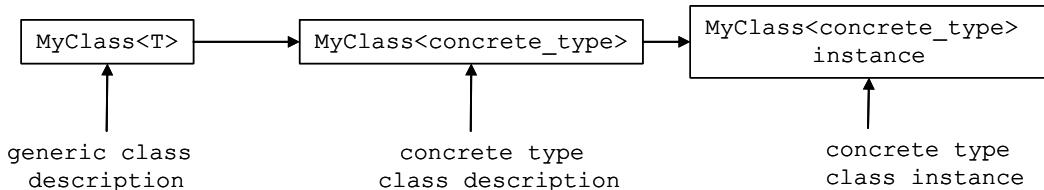
- As a method of unknown type parameter **T**:

```
public void Shelter(T newAnimal)
{
  // Method's body goes here …
}
```

- And a method, which returns a result of unknown type **T**:

```
public T Release(int i)
{
  // Method's body goes here …
}
```

As we already know when we create an object from our class shelter and replace the unknown type with a specific one (e.g. **Cat**), during the execution of the program, the above methods will have the following form:

- The parameter of method **Shelter** will be of type **Cat**:

```
public void Shelter(Cat newAnimal)
{
    // Method's body goes here …
}
```

- The method **Release** will return a result of type **Cat**:

```
public Cat Release(int i)
{
    // Method's body goes here …
}
```

Typifying (Generics) – Behind the Scenes

Before we continue, let's us explain what happens into the memory of the computer, when we work with generic classes.

First we declare our generic class **MyClass<T>** (generic class description in the scheme above). Then the compiler translates our code to an intermediate language (MSIL), as translated code contains information that the class is generic, i.e. it works with undefined types until now. At runtime, when someone tries to work with our generic class and tries to use it with a specific type, a new **description of the class** is created (specific type class description in the diagram above), which is identical to the generic class, with the difference that where it has been used **T**, now is replaced by a specific type. For example, if you try to use **MyClass<int>**, everywhere in your code, where the unknown parameter **T** is used, it will be replaced with **int**. Only then we can create object of a generic class with a specific type **int**. The interesting thing here is that to create this object, the description of the class, which was created in the meantime (specific type class description), will be used. Instantiating of a generic class by given specific types of its parameters is called "**specialization of the type**" or "**extension of generic class**".

Using our example, if we create an object of type **AnimalShelter<T>**, which works only with objects of type **Dog**, if we try to add an object of type **Cat**, this will cause a compile error almost identical to the errors, that were derived by an attempt to add an object of type **Cat**, into an object of type **AnimalShelter**, which we have created in section "Shelter for Homeless Animals – Example":

```
static void Main()
{
  AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>(10);

  Cat cat1 = new Cat();

  dogsShelter.Shelter(cat1);
}
```

As expected, we get the following **compilation error messages**:

```
The best overloaded method match for 'AnimalShelter<
Dog>.Shelter(Dog)' has some invalid arguments

Argument 1: cannot convert from 'Cat' to 'Dog'
```

Generic Methods

Like classes, when the type of method’s parameters cannot be specified, we can **parameterize (typify) the method**. Accordingly, the indication of a specific type will happen during the invocation of the method, replacing the unknown type with a specific one, as we did in the classes.

Typifying of a method is done, when after the name and before the opening bracket of the method, we add **<K>**, where **K** is the replacement of the type that will be used later:

```
<return_type><methods_name><K>(<params>)
```

Accordingly, we can use unknown type **K** for parameters in the parameter’s list of method **<params>**, whose type is unknown and also for return value or to declare variables of type substitute **K** in the body of the method.

For example, consider a **method that swaps the values of two variables**:

```
public void Swap<K>(ref K a, ref K b)
{
  K oldA = a;
  a = b;
  b = oldA;
}
```

This is a method that swaps the values of two variables, **without carrying of their types**. That is why we define it as a generic, so we can use it for all types of variables.

Accordingly, if we want to swap the values of two integers and then two string variables, we should use our method:

```
int num1 = 3;
int num2 = 5;
Console.WriteLine("Before swap: {0} {1}", num1, num2);
// Invoking the method with concrete type (int)
Swap<int>(ref num1, ref num2);
Console.WriteLine("After swap: {0} {1}\n", num1, num2);

string str1 = "Hello";
string str2 = "There";
Console.WriteLine("Before swap: {0} {1}!", str1, str2);
// Invoking the method with concrete type (string)
Swap<string>(ref str1, ref str2);
Console.WriteLine("After swap: {0} {1}!", str1, str2);
```

When you run this code, the result is as expected:

```
Before swap: 3 5
After swap: 5 3

Before swap: Hello There!
After swap: There Hello!
```

We notice that in the list of parameters we have used also the keyword **ref**. This concerns the specification of the method – namely, to exchange the values of two references. By using the keyword **ref**, the method will use the same reference that was given by the calling method. This way, all changes on this variable made by our method, will remain after the method exits.

We should know that by **calling a generic method**, we can miss the explicit declaration of a specific type (in our example **<int>**), because the compiler will detect it automatically, recognizing the type of the given parameters. In other words, our code can be simplified using the following calls:

```
Swap(ref num1, ref num2); // Invoking the method Swap<int>
Swap(ref str1, ref str2); // Invoking the method Swap<string>
```

We should know that the **compiler will be able to recognize what is the specific type**, only if this type is involved in the parameter's list. The compiler cannot recognize what is the specific type of a generic method only by the type its return value or if it does not have parameters. In both cases, this specific type will have to be given explicitly. In our example, it will be similar to the original method call, or by adding **<int>** or **<string>**.

It should be noticed that static methods can also be typified, unlike the properties and constructors of the class.

	Static methods can also be typified, but properties and constructors of the class cannot.

Features by Declaration of Generic Methods in Generic Classes

As we have already seen in the section "Using Unknown Types in a Declaration of Methods", non-generic methods can use unknown types, described in the generic class declaration (e.g. methods **Shelter()** and **Release()** from the example "Shelter for Homeless Animals"):

AnimalShelter.cs

```
public class AnimalShelter<T>
{
   // … The rest of the code …

   public void Shelter(T newAnimal)
   {
      // Method body here
   }

   public T Release(int i)
   {
      // Method body here
   }
}
```

If we try to reuse the variable, which is used to mark the unknown type of the generic class, for example as **T**, in the declaration of generic method, then when we try to compile the class, we will get a warning **CS0693**. This is happening because the scope of action of the unknown type **T**, defined in declaration of the method, overlaps the scope of action of the unknown type **T**, in class declaration:

CommonOperations.cs

```
public class CommonOperations<T>
{
   // CS0693
   public void Swap<T>(ref T a, ref T b)
   {
```

```
        T oldA = a;
        a = b;
        b = oldA;
    }
}
```

When you try to compile this class, you receive the following **message**:

```
Type parameter 'T' has the same name as the type parameter from
outer type 'CommonOperations<T>'
```

So if we want our code to be flexible, and our generic method safely to be called with a specific type, different from that in the generic class by instantiating it, we just have to declare the replacement of the unknown type in the declaration of the generic method **to be different than the parameter for the unknown type** in the class declaration, as shown below:

CommonOperations.cs

```
public class CommonOperations<T>
{
    // No warning
    public void Swap<K>(ref K a, ref K b)
    {
        K oldA = a;
        a = b;
        b = oldA;
    }
}
```

Thus, always make sure that there will be no overlapping of substitutes of the unknown types of method and class.

Using a Keyword "default" in a Generic Source Code

Once we have introduced the basics of generic types, let's try to **redesign our first example** in this section (Shelter for Homeless Animals). The only thing we need to do is to replace the type **Dog** with some parameter **T**:

AnimalsShelter.cs

```
public class AnimalShelter<T>
{
    private const int DefaultPlacesCount = 20;

    private T[] animalList;
```

```
    private int usedPlaces;

    public AnimalShelter() : this(DefaultPlacesCount)
    {
    }

    public AnimalShelter(int placesCount)
    {
        this.animalList = new T[placesCount];
        this.usedPlaces = 0;
    }

    public void Shelter(T newAnimal)
    {
        if (this.usedPlaces >= this.animalList.Length)
        {
            throw new InvalidOperationException("Shelter is full.");
        }
        this.animalList[this.usedPlaces] = newAnimal;
        this.usedPlaces++;
    }

    public T Release(int index)
    {
        if (index < 0 || index >= this.usedPlaces)
        {
            throw new ArgumentOutOfRangeException(
                "Invalid cell index: " + index);
        }
        T releasedAnimal = this.animalList[index];
        for (int i = index; i <this.usedPlaces - 1; i++)
        {
            this.animalList[i] = this.animalList[i + 1];
        }
        this.animalList[this.usedPlaces - 1] = null;
        this.usedPlaces--;

        return releasedAnimal;
    }
}
```

Everything looks to work properly, until we try to compile the class. Then we **get the following error**:

```
Cannot convert null to type parameter 'T' because it could be a
non-nullable value type. Consider using 'default(T)' instead.
```

The error is inside the method **Release()** and it is related to the recording a **null** value in the last released (rightmost) cell in the shelter. The problem is that we are trying to use the default value for a reference type, but **we are not sure whether this type is a reference type or a primitive**. Therefore the compiler displays the errors above. If the type **AnimalShelter** is instantiated by a structure and not by a class, then the null value is not valid.

To handle this problem, in our code we have to use the construct **default(T)** instead of **null**, which returns the default value for the particular type that will be used instead of **T**. As we know, the default value for reference type is **null**, and for numeric types – zero. We can make the following change:

```
// this.animalList[this.usedPlaces - 1] = null;
this.animalList[this.usedPlaces - 1] = default(T);
```

Finally the compilation runs smoothly and the class **AnimalShelter<T>** operates correctly. We can test it as follows:

```
static void Main()
{
  AnimalShelter<Dog> shelter = new AnimalShelter<Dog>();
  shelter.Shelter(new Dog());
  shelter.Shelter(new Dog());
  shelter.Shelter(new Dog());
  Dog d = shelter.Release(1); // Release the second dog
  Console.WriteLine(d);
  d = shelter.Release(0); // Release the first dog
  Console.WriteLine(d);
  d = shelter.Release(0); // Release the third dog
  Console.WriteLine(d);
  d = shelter.Release(0); // Exception: invalid cell index
}
```

Advantages and Disadvantages of Generics

Generic classes and methods **increase the reusability of the code**, the security and the performance compared to other non-generic alternatives.

As a general rule, the **programmer should strive to create and use generic classes, whenever it is possible**. The more generic types are used, the higher level of abstraction there is in the program and the source code becomes more flexible and reusable. However we should keep in mind, that overuse of generics can lead to over-generalization and the code may become unreadable and difficult to understand by other programmers.

Naming the Parameters of the Generic Types

Before we finish generics as a topic, let's give you some guidance on working with the substitutes (parameters) of unknown types in a generic class:

1. If there is just one unknown type in the generic, it is common to use the letter **T**, as a substitute for that unknown type. As an example we can give our class declaration **AnimalShelter<T>**, which we used until now.

2. To the substitutes should be given the most descriptive names, unless a letter is not a sufficiently descriptive and well-chosen name, this will not improve readability of the source code. For instance, we can modify our example, replacing the letter **T**, with the more descriptive substitute for **Animal**:

AnimalShelter.cs

```
public class AnimalShelter<Animal>
{
  // … The rest of the code …

  public void Shelter(Animal newAnimal)
  {
    // Method body here
  }

  public Animal Release(int i)
  {
    // Method body here
  }
}
```

When we use descriptive names of substitutes instead of a letter, it is better to add **T** at the beginning of the name, to distinguish it more easily from the class names in our application. In other words, instead of using a substitute **Animal** in the previous example, we should use **TAnimal** (**T** comes from the word "template" which means a parameterized / generic type).

Exercises

1. Define a class **Student**, which contains the following **information about students**: full name, course, subject, university, e-mail and phone number.

2. Declare several **constructors** for the class **Student**, which have different lists of parameters (for complete information about a student or part of it). Data, which has no initial value to be initialized with **null**. Use nullable types for all non-mandatory data.

3. Add a **static field** for the class **Student**, which holds the number of created objects of this class.
4. Add a **method** in the class **Student**, which displays complete information about the student.
5. Modify the current source code of **Student** class so as to **encapsulate** the data in the class using **properties**.
6. Write a class **StudentTest**, which has to **test the functionality** of the class **Student**.
7. Add a **static method** in class **StudentTest**, which creates several objects of type **Student** and store them in static fields. Create a **static property** of the class to access them. Write a test program, which displays the information about them in the console.
8. Define a class, which contains information about a **mobile phone**: model, manufacturer, price, owner, features of the battery (model, idle time and hours talk) and features of the screen (size and colors).
9. Declare several **constructors** for each of the classes created by the previous task, which have different lists of parameters (for complete information about a student or part of it). Data fields that are unknown have to be initialized respectively with **null** or **0**.
10. To the class of mobile phone in the previous two tasks, add a **static field nokiaN95**, which stores information about mobile phone model Nokia N95. Add a method to the same class, which displays information about this static field.
11. Add an **enumeration BatteryType**, which contains the values for type of the battery (Li-Ion, NiMH, NiCd, …) and use it as a new field for the class **Battery**.
12. Add a method to the class **GSM**, which returns information about the object as a **string**.
13. Define properties to encapsulate the data in classes **GSM**, **Battery** and **Display**.
14. Write a class **GSMTest**, which has to **test the functionality** of class **GSM**. Create few objects of the class and store them into an array. Display information about the created objects. Display information about the static field **nokiaN95**.
15. Create a class **Call**, which contains information about a call made via mobile phone. It should contain information about date, time of start and duration of the call.
16. Add a property for keeping a **call history – CallHistory**, which holds a list of call records.

17. In **GSM** class add methods for adding and deleting calls (**Call**) in the archive of mobile phone calls. Add method, which deletes all calls from the archive.

18. In **GSM** class, add a method that calculates the total amount of calls (**Call**) from the archive of phone calls (**CallHistory**), as the price of a phone call is passed as a parameter to the method.

19. Create a class **GSMCallHistoryTest**, with which to test the functionality of the class **GSM**, from task 12, as an object of type **GSM**. Then add to it a few phone calls (**Call**). Display information about each phone call. Assuming that the price per minute is 0.37, calculate and display the total cost of all calls. Remove the longest conversation from archive with phone calls and calculate the total price for all calls again. Finally, clear the archive.

20. There is a **book library**. Define classes respectively for a **book** and a **library**. The library must contain a name and a list of books. The books must contain the title, author, publisher, release date and ISBN-number. In the class, which describes the library, create methods to add a book to the library, to search for a book by a predefined author, to display information about a book and to delete a book from the library.

21. Write a **test class**, which creates an object of type library, adds several books to it and displays information about each of them. Implement a test functionality, which finds all books authored by Stephen King and deletes them. Finally, display information for each of the remaining books.

22. We have a **school**. In school we have **classes** and **students**. Each class has a number of **teachers**. Each teacher has a variety of disciplines taught. Students have a name and a unique number in the class. Classes have a unique text identifier. Disciplines have a name, number of lessons and number of exercises. The task is to shape a school with C# classes. You have to define classes with their fields, properties, methods and constructors. Also **define a test class**, which demonstrates, that the other classes work correctly.

23. Write a **generic class GenericList<T>**, which holds a list of elements of type **T**. Store the list of elements into an array with a limited capacity that is passed as a parameter of the constructor of the class. Add methods to add an item, to access an item by index, to remove an item by index, to insert an item at given position, to clear the list, to search for an item by value and to override the method **ToString()**.

24. Implement **auto-resizing functionality** of the array from the previous task, when by adding an element, it reaches the capacity of the array.

25. Define a class **Fraction**, which contains information about the **rational fraction** (e.g. ¼ or ½). Define a static method **Parse()** to create a fraction from a sting (for example **-3/4**). Define the appropriate

properties and constructors of the class. Also write property of type **Decimal** to return the decimal value of the fraction (e.g. 0.25).

26. Write a class **FractionTest**, which tests the functionality of the class **Fraction** from previous task. Pay close attention on testing the function Parse with different input data.

27. Write a function to **cancel a fraction** (e.g. if numerator and denominator are respectively 10 and 15, fraction to be cancelled to 2/3).

Solutions and Guidelines

1. Use **enum** for subjects and universities.

2. To avoid repetition of source code call **constructors** from each other with keyword **this(<parameters>)**.

3. Use the constructor of the class as a place where the number of objects of class **Student** is increasing.

4. Display on the console in all fields of the class **Student**, followed by a blank line.

5. Define as **private** all members of the class **Student** and then using Visual Studio (Refactor -> Encapsulate Field) define automatically the public **get** / **set** methods to access these fields.

6. **Create a few students** and display the whole information for each one of them.

7. You can use the **static constructor** to create instances in the first access to the class.

8. Declare three separate classes: **GSM**, **Battery** and **Display**.

9. Define the described constructors and **create a test program** to check if classes are working properly.

10. Define a **private** field and initialize it at the time of its declaration.

11. Use **enum** for the **type of battery**. Search in Internet for other types of batteries for phones, except these in the requirements and add them as value of the enumeration.

12. Override the method **ToString()**.

13. In classes **GSM**, **Battery** and **Display** define suitable **private** fields and generate **get** / **set**. You can use automatic generation in Visual Studio.

14. Add a method **PrintInfo()** in class **GSM**.

15. Read about the class **List<T>** in Internet. The class **GSM** has to store its conversations in a list of type **List<Call>**.

16. Return as a result the **list of conversations**.

17. Use the built-in methods of the class **`List<T>`**.
18. Because the **tariff is fixed**, you can easily **calculate the total price** of all calls.
19. **Follow the instructions** directly from the requirements of the task.
20. Define classes **`Book`** and **`Library`**. For a list of books use **`List<Book>`**.
21. Follow the instructions directly from the requirements of the task.
22. Create classes **`School`**, **`SchoolClass`**, **`Student`**, **`Teacher`**, **`Discipline`** and define into them their respective fields, as described in the instructions of the task. Do not use the word "**`Class`**" as a class name, because in C# it has special meaning. Add methods for printing all the fields from each of the classes.
23. Use your knowledge concerning **generic classes**. Check out all input parameters of the methods, just to make sure that no element can access an invalid position.
24. When you reach the capacity of the array, **create a new array with a double size** and copy all old elements in the new one.
25. Write a class with two **`private decimal`** fields, which hold information relevant to the **numerator** and **denominator** of the fraction. Among other requirements in the task, redefine in appropriate standard the features for each object: **`Equals(…)`**, **`GetHashCode()`**, **`ToString()`**.
26. Figure out appropriate **tests**, for which your function may give incorrect results. Good practice is **first to write the tests**, then to implement their specific functionality.
27. Search for information in Internet for the "**greatest common divisor (GCD)**" and the **Euclidean algorithm** for its calculation. Divide the numerator and denominator of their greatest common divisor and you will get the cancelled fraction.

Chapter 15. Text Files

In This Chapter

In this chapter we will review how to **work with text files** in C#. We will explain what a **stream** is, what its purpose is, and how to use it. We will explain what a text file is and how can you **read and write data to a text file** and how to deal with different **character encodings**. We will demonstrate and explain the good practices for exception handling when working with files. All of this will be demonstrated with many examples in this chapter.

Streams

Streams are an **essential part of any input-output library**. You can use streams when your program needs to "read" or "write" data to an external data source such as files, other PCs, servers etc. It is important to say that the term **input** is associated with reading data, whereas the term **output** is associated with writing data.

What Is a Stream?

A **stream** is an **ordered sequence of bytes**, which is send from one application or input device to another application or output device. These bytes are written and read one after the other and always arrive in the same order as they were sent. Streams are an **abstraction of a data communication channel that connects two devices or applications**.

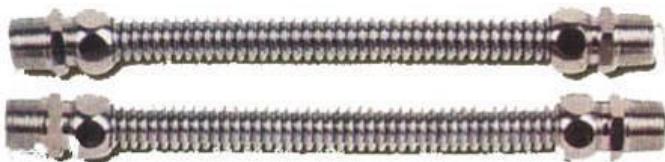
Streams are the primary means of exchanging information in the computer world. Because of streams, different applications are able to access files on the computer and are able to establish network communication between remote computers. In the world of computers, many operations can be interpreted as **reading and writing to a stream**. For example, printing is a process of sending a sequence of bytes to a stream, associated with the corresponding port, to which is the printer connected. Recreating sounds from the computer's sound card can be done by sending some commands, followed by the sample sound, which is actually a sequence of bytes. The scanning of documents from a scanner can be done by sending commands to the scanner (an output stream) and then reading the scanned image (an input stream). This way, you can work with any peripheral device (camera, mouse, keyboard, USB stick, soundcard, printer, scanner etc.).

Every time when you read or write from or to a file, you have to **open a stream** to the corresponding file, **do the reading or writing**, and then

close the stream. There are two types of streams – **text streams** and **binary streams** but this separation has to do with the interpretation of the sent and received bytes. Sometimes, for convenience, a sequence of bytes can be treated as text (in a predefined encoding) and is referred to as a text stream.

Today's modern web sites cannot do without the so-called **streaming**, which represents stream access to bulky multimedia files coming from the Internet. Streaming audio and video allows files to be played before they are downloaded locally, making the site more interactive. Streams and media streaming are different concepts but both use **sequences of data**.

Basic Things You Need to Know about Streams

Many devices **use streams for reading and writing** data. Because of streams, communication between program and file, program and remote computer, is made easy.

Streams are **ordered sequences of bytes**. The word "order" is intentionally left stressed, because it is of great importance to remember that streams are highly ordered and organized. In no way must you influence the order of the information flow, because it will render it unusable. If a byte is sent to a stream earlier than another byte, it will arrive earlier at the other end of the stream, which is guaranteed by the abstraction "stream".

Streams allow **sequential data access**. Again, it is important to understand the meaning of the word sequential. You can manipulate the data only in the order in which it arrives from the stream. This is closely related to the previous feature. You cannot take the first, than the eight, third, thirteenth byte and so on. Streams **do not allow random access** to their data, only **sequential**. You can think of streams as of a linked list that contains bytes, in which they have a strict order.

Different situations require **different types of streams**. Some streams are used with text files, others-with binary files and then there are those that work with strings. For network communication, you have to use a specific type of stream. The vast variety of streams can help us in different situations, but can also trouble us, because we need to be familiar with every type of stream, before we can use it in our application.

Streams are opened before we can begin working with them and are closed after they have served their purpose. **Closing the stream is very important** and must not be left out, because you risk losing data, damaging the file, to which the stream is opened, and so on – all of these are very troublesome scenarios, which must not happen in our programs.

We can say that streams are like **pipes that connect two points**:

From one side **we pour data in** and from the other **data leaks out**. The one who pours data is not concerned of how it is transferred, but can be sure that what he has poured will come out the same on the other side. Those who use streams do not care how the data reaches them. They know that if someone poured something on the other side, it will reach them. Therefore, we can consider streams as a **data transport channel**, such as pipes.

Basic Operations with Streams

You can do the following **operations with streams**: creation / opening, reading data, writing data, seeking / positioning, closing / disconnecting.

Creation

To **create or open a stream** means to connect the stream to a data source, a mechanism for data transfer or another stream. For example, when we have a file stream, then we pass the file name and the file mode in which it is to be opened (reading, writing or reading and writing simultaneously).

Reading

Reading means extracting data from the stream. Reading is always performed **sequentially** from the current position of the stream. Reading is a **blocking operation**, and if the other party has not sent data while we are trying to read or the sent data has not yet arrived, there may occur a delay – a few milliseconds to hours, days or greater. For example, when reading from a network stream data can be slowed down because of the network or the other party might not have send any data.

Writing

Writing means sending data to the stream in a specific way. The writing is performed from the current position of the stream. Writing may be a potentially **blocking operation**, before the data is sent on its way. For example, if you send bulk data via a network stream, the operation may be delayed while the data is traveling over the network.

Positioning

Positioning or seeking in the stream means to move the current position of the stream. Moving is done according to the current position, where we can position according to the current position, beginning of the stream, or the end of the stream. Moving can be done only in **streams that support positioning**. For example, file streams typically maintain positioning while network streams do not.

Closing

To close or disconnect a stream means to complete the work with the stream and **releases the occupied resources**. Closing must take place as soon as possible after the stream has served its purpose, because a resource opened by a user, usually cannot be used by other users (including other programs on the same computer that run parallel to our program).

Streams in .NET – Basic Classes

In .NET Framework **classes for working with streams** are located in the namespace **System.IO**. Let's focus on their hierarchy, organization and functionality.

We can distinguish two main types of streams – those who work with **binary data** and those who work with **text data**. Later we will discuss the main characteristics of these two types.

At the top of the stream hierarchy stands an **abstract input-output stream class**. It cannot be instantiated, but defines the basic functionality that all the other streams have.

There are **buffered streams** that do not add any extra functionality, but use a buffer for reading and writing data, which significantly enhances performance. Buffered streams will not be analyzed in this chapter, as we will focus on working with text files. You can check with the rich documentation available on the Internet or a textbook for advanced programming.

Some streams **add additional functionality** to reading and writing data. For example, there are streams that compress / decompress data sent to them and streams that encrypt / decrypt data. These streams are connected to another stream (such as file or network stream) and add additional processing to its functionality.

The main classes in the **System.IO** namespace are **Stream** (abstract base class for all streams in .NET Framework), **BufferedStream**, **FileStream**, **MemoryStream**, **GZipStream** and **NetworkStream**. We will discuss in more details some of them, separating them in their basic feature – the type of data with which they work.

All streams in C# are similar in one basic thing – **it is mandatory to close them** after we have finished working with them. Otherwise we risk damaging the data in the stream or file that we have opened. This brings us to the first and basic rule that we should always remember when working with streams:

	Always close the streams and files you work with! Leaving an open stream or file leads to loss of resources and can block the work of other users or processes in your system.

Binary and Text Streams

As we mentioned earlier, we can divide the streams into two large groups according to the type of data that we deal with – **binary streams** and **text streams**.

Binary Streams

Binary streams, as their name suggests, work with **binary (raw) data**. You probably guess that that makes them universal and they can be used to read information from all sorts of files (images, music and multimedia files, text files etc.). We will take a brief look over them, because we will currently focus on working with text files.

The main classes that we use to read and write from and to binary streams are: **FileStream**, **BinaryReader** and **BinaryWriter**.

The class **FileStream** provides us with various methods for **reading and writing from a binary file** (read / write one byte and a sequence of bytes), skipping a number of bytes, checking the number of bytes available and, of course, a method for closing the stream. We can get an object of that class by calling him his constructor with parameter-a file name.

The class **BinaryWriter** enables you to **write primitive types and binary values** in a specific encoding to a stream. It has one main method – **Write(…)**, which allows recording of any primitive data types – integers, characters, Booleans, arrays, strings and more.

BinaryReader allows you to **read primitive data types and binary values** recorded using a **BinaryWriter**. Its main methods allow us to read a character, an array of characters, integers, floating point, etc. Like the previous two classes, we can get on object of that class by calling its constructor.

Text Streams

Text streams are very similar to binary, but only **work with text data** or rather a sequence of characters (**char**) and strings (**string**). Text streams are ideal for working with text files. On the other hand, this makes them unusable when working with any binaries.

The main classes for working with text streams in .NET are **TextReader** and **TextWriter**. They are abstract classes, and they cannot be instantiated. These classes define the basic functionality for reading and writing for the classes that inherit them. Their more important methods are:

- **ReadLine()** – reads one line of text and returns a string.
- **ReadToEnd()** – reads the entire stream to its end and returns a string.
- **Write()** – writes a string to the stream.
- **WriteLine()** – writes one line of text into the stream.

As you know, the characters in .NET are Unicode characters, but **streams can also work with Unicode and other encodings** like the standard encoding for Cyrillic languages Windows-1251.

The classes, to which we will turn our attention to in this chapter, are **StreamReader** and **StreamWriter**. They directly inherit the **TextReader** and **TextWriter** classes and implement functionality for **reading and writing textual information to and from a file**.

To create an object of type **StreamReader** or **StreamWriter**, we need a file or a string, containing the file path. Working with these classes, we can use all of the methods that we are already familiar with, to work with the console. Reading and writing to the console is much like reading and writing respectively with **StreamReader** and **StreamWriter**.

Relationship between Text and Binary Streams

When writing text, hidden from us, the class **StreamWriter transforms the text into bytes** before recording it at the current position in the file. For this purpose, it uses the character encoding, which is set during its creation. The **StreamReader** class works similarly. It uses **StringBuilder** internally and when reading binary data from a file, it **converts the received bytes to text** before sending the text back as a result from reading.

Remember that the **operating systems have no concept of "text file"**. The file is always a sequence of bytes, but whether it is text or binary depends on the interpretation of these bytes. If we want to look at a file or a stream as text, we must read and write to it with text streams (**StreamReader** or **StreamWriter**), but if we wish to treat it as binary, we must read and write with a binary stream (**FileStream**).

Bear in mind that text streams work with text lines, that is, they **interpret binary data as a sequence of text lines**, separated from each other with a new line separators.

The **character for the new line** is not the same for different platforms and operating systems. For UNIX and Linux it is **LF (0x0A)**, for Windows and DOS it is **CR + LF (0x0D + 0x0A)**, and for Mac OS (up to version 9) it is **CR (0x0A)**. Reading one line of text from a given file or a stream means reading a sequence of bytes until reading one of the characters **CR** or **LF** and converting these bytes to text according to the encoding, used by the stream. Similarly, writing one line of text to a text file or stream means writing the binary representation of the text (according to the current encoding), followed by the character (or characters) for a new line for the current operating system (such as **CR + LF**).

Reading from a Text File

Text files provide the ideal solution for reading and writing data. If we want to enter some data automatically (instead by hand), we could read it from a text

files. So now, we will take a look at how to read and write text files with the classes from .NET Framework and the C# language.

StreamReader Class for Reading a Text File

C# provides several ways to read files but not all are easy and intuitive to use. This is why we will use the **StreamReader** class. The **System.IO. StreamReader** class provides the easiest way to read a text file, as it resembles reading from the console, which by now you have probably mastered to perfection.

Having read everything until now, you are probably a bit confused. We already explained that reading and writing to and from text files is only and exclusively possible with streams, but **StreamReader** did not appear anywhere in the above-mentioned streams and you are not sure whether it is actually a stream. Indeed, **StreamReader** is not a stream, but it can work with streams. It provides the easiest and comprehensive way to read from a text file.

Opening a Text File for Reading

You can simply create a **StreamReader** from a filename (or full file path), which greatly eases us and reduces the probability of an error. On its creation, we can specify the character encoding. Here is an example of how an object of the class **StreamReader** can be created:

```
// Create a StreamReader connected to a file
StreamReader reader = new StreamReader("test.txt");

// Read the file here …

// Close the reader resource after you've finished using it
reader.Close();
```

The first thing to do, when reading from a text file, is to create a variable of type **StreamReader**, which we can associate with a specific file from the file system on our computer. To do this we need only pass the file path as a parameter to the constructor. Note that if the file is located in the folder where the compiled project (subdirectory **bin\Debug**) is, we can only provide its filename. Otherwise, we have to provide the full file path or relative path.

The code in the above example that creates an object of type **StreamReader** can cause an error. For now, simply pass a path to an existing file, and later on we will turn to the handling of errors when working with files.

Full and Relative Paths

When working with files we can **use full paths** (e.g. **C:\Temp\example.txt**) or **relative paths**, to the directory from which the application was started (e.g. **..\..\example.txt**).

If you use full paths, where you pass the full path to a file, do not forget to apply escaping of slashes, which is used to separate the folders. In C# you can do this in two ways – with a double slash or with a quoted string beginning with **@** before the string literal. For example, to enroll the path to the file "**C:\Temp\work\test.txt**" in a string we have two options:

```
string fileName = "C:\\Temp\\work\\test.txt";
string theSamefileName = @"C:\Temp\work\test.txt";
```

Although the use of relative paths is more difficult because you have to take into account the directory structure of your project which may change during the life of the project, it is **highly recommended avoiding full paths**.

Avoid full file paths and work with relative paths! This makes your application portable and easy for installation and maintenance.

Using the full path to a file (e.g. **C:\Temp\test.txt**) is bad practice because it makes your application **dependent on the environment** and also non-transferable. If you transfer it to another computer, you will need to correct paths to the files, which it seeks, to work correctly. If you use a relative path to the current directory (e.g. **..\..\example.txt**), your program will be easily portable.

Remember that when you start the C# program, the current directory is the one, in which the executable (.exe) file is located. Most often this is the subdirectory bin\Debug or bin\Release directory to the root of the project. Therefore, to open the file example.txt from the root directory of your Visual Studio project, you should use a relative path ..\..\example.txt.

Universal Relative to Physical Path Resolver

If you want to write a portable application, you might benefit of Nakov's universal path resolver: http://www.nakov.com/blog/2009/07/14/universal-relative-to-physical-path-resolver-for-console-wpf-and-aspnet-apps/. It can automatically **resolve a relative path to full (physical) file path** in Web, desktop, console or other .NET application. For example, if your application consists of an assembly **App.exe** and a file **logo.gif** and these files are located in the same directory, at runtime you will be able to get the physical location of **logo.gif** through the following code:

```
string logoPath =
  UniversalFilePathResolver.ResolvePath(@"~\logo.gif");
```

Reading a Text File Line by Line – Example

Now, we have learned how to create **StreamReader**. We can go further by trying to do something more complicated: to **read the entire text file line by line** and print the read text on to the console. Our advice is to create the text file in the **Debug** folder of the project (**.\bin\Debug**), so that it will be in the same directory in which your compiled application will be and you will not have to set the full path to it when opening the file. Let's see what our file looks like:

Sample.txt

```
This is our first line.
This is our second line.
This is our third line.
This is our fourth line.
```

We have a text file from which to read. Now we must create an object of type **StreamReader** to read the file and loop though it line by line:

FileReader.cs

```
class FileReader
{
  static void Main()
  {
    // Create an instance of StreamReader to read from a file
    StreamReader reader = new StreamReader("Sample.txt");

    int lineNumber = 0;

    // Read first line from the text file
    string line = reader.ReadLine();

    // Read the other lines from the text file
    while (line != null)
    {
      lineNumber++;
      Console.WriteLine("Line {0}: {1}", lineNumber, line);
      line = reader.ReadLine();
    }

    // Close the resource after you've finished using it
    reader.Close();
  }
}
```

There is nothing difficult to read text files. The first part of our program is already well known – create a variable of type **StreamReader**, to whose constructor we pass the file's name, which will be read. The parameter of the constructor is the path to the file, but since our file is found in the **Debug** directory of the project, we set only its name as a path. If our file were located in the project directory, then we would have submitted the string – **"..\..\Sample.txt"** as a path.

After that, we create a variable – counter, whose purpose is to count and display on which row of the file we are currently located.

Then, we create a variable that will store each read line. With its creation, we directly read the first line of text file. If the text file is empty, the method **ReadLine()** of the **StreamReader** object will return **null**.

For the main part – reading the file line by line, we will use a **while** loop. The condition for the loop is: as long as there is something in the variable **line**, we should continue reading. In the body of the loop, our task is to increase the value of the counter variable by one and then print the current line in the format we like. Finally, again we use **ReadLine()** to read the next line in the file and write it in the variable **line**. For printing, we use a method that is well known to us from the tasks, which required something to be printed on to the console – **WriteLine()**.

Once we have read everything we need from the file, we should not forget to close the object **StreamReader**, as to avoid loss of resources. For this, we use the method **Close()**.

Always close the StreamReader instances after you finish working with them. Otherwise you risk losing system resources. Use the method Close() or the statement using.

The result of the program should look like this:

```
Line 1: This is our first line.
Line 2: This is our second line.
Line 3: This is our third line.
Line 4: This is our fourth line.
```

Automatic Closing of the Stream after Working with It

As noted in the previous example, having finished working with the object of type **StreamReader**, we called **Close()** and closed the stream behind the **StreamReader** object. Very often, however, novice programmers forget to call the **Close()** method, thus blocking the file they use. Also in case of runtime exception when reading from a file, the file might be left open. This causes resource leakage and can lead to very unpleasant effects like program hanging, program misbehavior and strange errors.

The correct way to handle the file closing is though the **using** keyword:

```
using (<stream object>) { … }
```

The C# construct **using(…)** ensures that after leaving its body, the method **Close() will automatically be called**. This will happen even if an exception occurs when reading the file.

Now let's rework the previous example to benefit from the **using** construct:

FileReader.cs

```
class FileReader
{
    static void Main()
    {
        // Create an instance of StreamReader to read from a file
        StreamReader reader = new StreamReader("Sample.txt");

        using (reader)
        {
            int lineNumber = 0;

            // Read first line from the text file
            string line = reader.ReadLine();

            // Read the other lines from the text file
            while (line != null)
            {
                lineNumber++;
                Console.WriteLine("Line {0}: {1}", lineNumber, line);
                line = reader.ReadLine();
            }
        }
    }
}
```

Now the code guarantees that once opened successfully, **the text file will be closed correctly** regardless of whether reading from it will succeed or fail.

If you are wondering how it is best to take care of closing your program's streams and files, follow the following rule:

Always use the using construct in C# in order to properly close files and streams!

File Encodings. Reading in Cyrillic

Let’s now consider the problems that occur when reading a file using an incorrect encoding, such as reading a file in Cyrillic.

Character Encodings

You know that in memory **everything is stored in binary form**. This means that it is necessary for text files to be represented digitally, so that they can be stored in memory, as well as on the hard disk. This process is called **encoding files** or more correctly encoding the characters stored in text files.

The **encoding process** consists of replacing the text characters (letters, digits, punctuation, etc.) with specific sequences of binary values. You can imagine this as a large table in which each character corresponds to a certain value (sequence of bytes).

We already know **the concept of character encodings** and few character encoding schemes like **UTF-8** and **Windows-1251** from the section "Encoding Schemes" of chapter "Numeral Systems and Data Representation" and also from the section about "File Encodings in Visual Studio" of chapter "Defining Classes". Now we will extend this concept a bit and will use character encodings to work correctly with text files.

Character encodings specify the rules for converting from text to sequence of bytes and vice versa. An encoding scheme is a table of characters along with their numbers, but may also contain special rules. For example, the character "accent" (U + 0300) is special and sticks to the last character that precedes it. It is encoded as one or more bytes (depending on the character encoding scheme), and it does not correspond to any character, but to a part of the character. We will take a look at two encodings that are used most often when working with Cyrillic: **UTF-8** and **Windows-1251**.

UTF-8 is a **universal encoding scheme**, which supports all languages and alphabets in the world. In UTF-8 the most commonly used characters (Latin alphabet, numerals and special characters) are encoded in one byte, rarely used Unicode characters (such as Cyrillic, Greek and Arabic) are encoded in two bytes and all other characters (Chinese, Japanese and many others) are encoded in 3 or 4 bytes. UTF-8 encoding can convert any Unicode text in binary form and back and support all of the 100,000 characters of Unicode standard. UTF-8 encoding is universal and suitable for any language alphabet.

Another commonly used encoding is **Windows-1251**, which is usually used for **encoding of Cyrillic texts** (such as messages sent by e-mail). It contains 256 characters, including the Latin alphabet, Cyrillic alphabet and some commonly used signs. It uses one byte for each character, but at the expense of some characters that cannot be stored in it (as the Chinese alphabet characters), and are lost in an attempt of doing so.

Other examples of encoding schemes (encodings or charsets) are **ISO 8859-1**, **Windows-1252**, **UTF-16**, **KOI8-R**, etc. They are used in specific regions of

the world and define their own sets of characters and rules for the transition from text to binary data and vice versa.

For working with encodings (charsets) in .NET Framework, the class **System.Text.Encoding** is used, which is created the following way:

```
Encoding win1251 = Encoding.GetEncoding("Windows-1251");
```

Reading a Cyrillic Content

You probably already guessed that if we want to read from a file that contains **characters from the Cyrillic alphabet**, we must use the correct encoding that "understands" correctly these special characters. Typically, in a Windows environment, text files, containing Cyrillic text, are stored in **Windows-1251** encoding. To use it, we should set it as the encoding of the stream, which our **StreamReader** will process:

```
Encoding win1251 = Encoding.GetEncoding("Windows-1251");
StreamReader reader = new StreamReader("test.txt", win1251);
```

If you do not explicitly set the encoding scheme (encoding) for the file read, in .NET Framework, the default encoding **UTF-8** will be used.

You might wonder what happens if you use wrong encoding when reading or writing a file. There are several scenarios possible:

- If you use read / write only Latin letters, everything will work normally.
- If you write Cyrillic letters, to a files open with encoding, which does not support the Cyrillic alphabet (e.g. **ASCII**), Cyrillic letters will be permanently replaced by the character "**?**" (question mark).

In any case, these are unpleasant problems, which cannot be immediately noticed.

To avoid problems with incorrect encoding of files, always check the encoding explicitly. Otherwise, you may work incorrectly or break at a later stage.

The Unicode Standard. Reading in Unicode

Unicode is an industry standard that allows computers and other electronic devices always to present and manipulate text, which was written in most of the world's literacies. It consists of over 100,000 characters, as well as various encoding schemes (encodings). The unification of different characters, which Unicode offers, leads to its greater distribution. As you know, characters in C# (types **char** and **string**) are also presented in Unicode.

To read characters, stored in Unicode, we must use one of the supported encoding schemes for this standard. The most popular and widely used is **UTF-8**. We can set it as a code scheme with an already familiar way:

```
StreamReader reader = new StreamReader("test.txt",
  Encoding.GetEncoding("UTF-8"));
```

If you are wondering, whether to read a text file, encoded in Cyrillic, **Windows-1251** or **UTF-8**, then this question has no clear answer. Both standards are widely used for the recording of non-Latin text. Both encoding schemes are allowed and can be used. You should only always follow the rule that a **certain files should always be read and written using the same encoding**.

Writing to a Text File

Text files are very convenient for storing various types of information. For example, we can record the results of a program. We can use text files to make something like a journal (log) for the program – a convenient way to monitor it at runtime.

Again, as with reading a text file, we will use a similar to the **Console** class when writing, called **StreamWriter**.

The StreamWriter Class

The class **StreamWriter** is part of the **System.IO** namespace and is used exclusively for working with text data. It resembles the class **StreamReader**, but instead of methods for reading, it offers similar methods for writing to a text file. Unlike other streams, before writing data to the desired destination, **StreamWriter** turns it into bytes. **StreamWriter** enables us to set a preferred character encoding at the time it is created. We can create an instance of the class the following way:

```
StreamWriter writer = new StreamWriter("test.txt");
```

In the constructor of the class can pass as a parameter a file path, as well as an existing stream, to which we will write, or an encoding scheme. The **StreamWriter** class has several predefined constructors, depending on whether we will write to a file or a stream. In the examples, we will use the constructor with the parameter – file path. Example of the usage of the **StreamWriter** class constructor with more than one parameter is:

```
StreamWriter writer = new StreamWriter("test.txt",
  false, Encoding.GetEncoding("Windows-1251"));
```

In this example, we pass a file path as the first parameter. As a second parameter, we pass a Boolean variable that indicates whether to overwrite the file or to append the data at the end of the file. As a third parameter, we pass an encoding scheme (charset).

The example lines of code could trigger an exception, but the handling of input / output exceptions will be discussed later in this chapter.

Printing the Numbers [1…20] in a Text File – Example

Once we know how to create a **StreamWriter** class, we will use it as intended. Our goal is to enroll in a text file the numbers from 1 to 20, each number on a separate line. We can do this the following way:

```
class FileWriter
{
  static void Main()
  {
    // Create a StreamWriter instance
    StreamWriter writer = new StreamWriter("numbers.txt");

    // Ensure the writer will be closed when no longer used
    using(writer)
    {
      // Loop through the numbers from 1 to 20 and write them
      for (int i = 1; i <= 20; i++)
      {
        writer.WriteLine(i);
      }
    }
  }
}
```

We start by creating an instance of **StreamWriter** in the already well-known way from the examples.

To list the numbers from 1 to 20 we will use a **for**-loop. Inside the loop, we use the method **WriteLine(…)**, which again we know from our previous work with the console, to record the current number on a new line in the file. You need not worry if a file with the chosen name does not exist. If such the case, it will automatically be created in the folder of the project and if it already exists, it will be overwritten (old content will be deleted). The outcome is:

numbers.txt
1 2 3 … 20

To make sure that after the end of the file it will be closed, we should use the **using** construct.

	Be sure to close the stream after you finish using it! The best way to dispose any unused resources is with the using construct in C#.

When you want to print text in Cyrillic and are unsure what encoding to use, prefer the UTF-8 encoding. It is universal and not only supports Cyrillic, but all widespread international alphabets: Greek, Arabic, Chinese, Japanese, etc.

Input / Output Exception Handling

If you have followed the examples so far, you have probably noticed that many of the operations, related to files, can cause exceptional situations. The basic principles and approaches for their capture and processing are already familiar to you from the chapter "Handling Exceptions". Now we will concentrate on the specific **errors when working with files** and best practices for their handling.

Intercepting Exceptions when Working with Files

Perhaps the most common exception when working with files is the **FileNotFoundException** (its name infers that the desired file was not found). It can occur when creating **StreamReader**.

When setting a specified encoding by the creation of a **StreamReader** or a **StreamWriter** object, an **ArgumentException** can be thrown. This means, that the encoding we have chosen is not supported.

Another common mistake is **IOException**. This is the base class for all IO errors when working with streams.

The standard approach for handling exceptions when working with files is the following: declare variables of class **StreamReader** or **StreamWriter** in **try-catch** block. Initialize them with the necessary values in the block and handle the potential exceptions properly. To close the stream, we use the structure **using**. To illustrate what we just said, will give an example.

Catching an Exception when Opening a File – Example

Here's how we can catch exceptions that occur when working with files:

```
class HandlingExceptions
{
   static void Main()
   {
      string fileName = @"somedir\somefile.txt";
      try
```

```
    {
      StreamReader reader = new StreamReader(fileName);
      Console.WriteLine(
        "File {0} successfully opened.", fileName);
      Console.WriteLine("File contents:");
      using (reader)
      {
        Console.WriteLine(reader.ReadToEnd());
      }
    }
    catch (FileNotFoundException)
    {
      Console.Error.WriteLine(
        "Can not find file {0}.", fileName);
    }
    catch (DirectoryNotFoundException)
    {
      Console.Error.WriteLine(
        "Invalid directory in the file path.");
    }
    catch (IOException)
    {
      Console.Error.WriteLine(
        "Can not open the file {0}", fileName);
    }
  }
}
```

The example demonstrates reading a file and printing its contents to the console. If we accidentally have confused the name of the file or have deleted it, an exception of type **FileNotFoundException** will be thrown. In the **catch** block we intercept this sort of exception and if such occurs, we will process it properly and print a message to the console, saying that this file cannot be found. The same will happen if there were no directory named **"somedir"**. Finally, for better security, we have also added a **catch** block for **IOExceptions**. There all other IO exceptions, that might occur when working with files, will be intercepted.

Text Files – More Examples

We hope the theoretical explanations and examples so far have helped you get into the subtleties when working with text files. Now we will take a look at some **more complex examples**, so as to review the gained knowledge and to illustrate how to use them in solving practical problems.

Occurrences of a Substring in a File – Example

Here is how to implement a simple program that counts how many times a substring occurs in a text file. In the example, let's look for the substring "**C#**" in a text file as follows:

sample.txt

```
This is our "Intro to Programming in C#" book.
In it you will learn the basics of C# programming.
You will find out how nice C# is.
```

We can implement the counting as follows: will read the file line by line and each time we meet the desired word inside the last read line, we will increase the value of a variable (counter). We will process the possible exceptional situations to enable users to receive adequate information in case of errors. Here is a sample implementation:

CountingWordOccurrences.cs

```
static void Main()
{
   string fileName = @"..\..\sample.txt";
   string word = "C#";
   try
   {
      StreamReader reader = new StreamReader(fileName);
      using (reader)
      {
         int occurrences = 0;
         string line = reader.ReadLine();
         while (line != null)
         {
            int index = line.IndexOf(word);
            while (index != -1)
            {
               occurrences++;
               index = line.IndexOf(word, (index + 1));
            }
            line = reader.ReadLine();
         }
         Console.WriteLine(
            "The word {0} occurs {1} times.", word, occurrences);
      }
   }
   catch (FileNotFoundException)
```

```
    {
        Console.Error.WriteLine(
            "Can not find file {0}.", fileName);
    }
    catch (IOException)
    {
        Console.Error.WriteLine(
            "Cannot read the file {0}.", fileName);
    }
}
```

For simplicity of the example, the word we seek is hardcoded. You can implement the program to search a word entered by the user.

You can see that the example is not very different from the previous ones. We initialize the variables outside of the **try-catch** block. Again, we use a **while**-loop to read the lines of the text file one by one. Inside its body, there is another while-loop, which counts how many times the searched word occurs in the given line, and then increases the number of occurrences. This is done using the method **IndexOf(…)** of the class **String** (remember what it does in case you have forgotten). We do not forget to ensure the closing of the **StreamReader** object using the **using** structure. All that remains is to print the results on to the console.

For our example, the result is the following:

```
The word C# occurs 3 times.
```

Editing a Subtitles File – Example

Now we will look at a more complex example, in which we at the same time read from a file and record to another. This program **fixes a subtitles file for a movie**.

Our goal will be to read a file with subtitles, that are incorrect and do not appear at the right time, and to shift the times in an appropriate manner, so that they can appear correctly. One such file generally contains the time of the on-screen duration and the text, that should appear in the defined time interval. Here is how **typical subtitles files** look like:

GORA.sub

```
{1029}{1122}{Y:i}Captain, systems are|at the ready.
{1123}{1270}{Y:i}The preassure is stable.|Prepare for landing.
{1343}{1468}{Y:i}Please,fasten your seatbelts|and take your
places.
{1509}{1610}{Y:i}Coordinates 5.6|- Five, Five, Six, dot com.
```

```
{1632}{1718}{Y:i}Where did the coordinates|go to?
{1756}{1820}Commander Logar,|everyone is speaking in English.
{1821}{1938}Can't we switch|to Turkish from the beginning?
{1942}{1992}Yes, we can!
{3104}{3228}{Y:b}G.O.R.A.|a movie about the cosmos
…
```

StarWars.sub

```
{1029}{1122}{Y:i}I'll never join you.
{1123}{1270}{Y:i}If you only knew | the power of the dark side.
{1343}{1468}{Y:i}Obi One never told you what happened to your father!
{1509}{1610}{Y:i}He told me enough! | He told me you killed him.
{1632}{1718}{Y:i}No... I am your father!
{1756}{1820}(dramatic music playing)...
{1821}{1938}No, no that's not true... | That's impossible!
{1942}{1992}Search your feelings,| you know it's true.
{3104}{3228}{Y:b}Nooo…
…
```

To **fix the subtitles**, we usually just need to make an adjustment in the time for displaying the subtitles. Such an adjustment may be offsetting the start / end time for each subtitle (by addition or subtraction of a constant) or changing the speed (multiplying by a factor, say 1.05).

Here is sample code that can implement such a program:

FixingSubtitles.cs

```
using System;
using System.IO;

class FixingSubtitles
{
  const double COEFFICIENT = 1.05;
  const int ADDITION = 5000;
  const string INPUT_FILE = @"..\..\source.sub";
  const string OUTPUT_FILE = @"..\..\fixed.sub";

  static void Main()
  {
    try
    {
```

```
        // Create reader
        StreamReader streamReader = new StreamReader(INPUT_FILE);

        // Create writer
        StreamWriter streamWriter =
            new StreamWriter(OUTPUT_FILE, false);

        using (streamReader)
        {
            using (streamWriter)
            {
                string line;
                while ((line = streamReader.ReadLine()) != null)
                {
                    streamWriter.WriteLine(FixLine(line));
                }
            }
        }
    }
    catch (IOException exc)
    {
        Console.WriteLine("Error: {0}.", exc.Message);
    }
}

static string FixLine(string line)
{
    // Find closing brace
    int bracketFromIndex = line.IndexOf('}');

    // Extract 'from' time
    string fromTime = line.Substring(1, bracketFromIndex - 1);

    // Calculate new 'from' time
    int newFromTime = (int) (Convert.ToInt32(fromTime) *
        COEFFICIENT + ADDITION);

    // Find the following closing brace
    int bracketToIndex = line.IndexOf('}',
        bracketFromIndex + 1);

    // Extract 'to' time
    string toTime = line.Substring(bracketFromIndex + 2,
        bracketToIndex - bracketFromIndex - 2);
```

```
        // Calculate new 'to' time
        int newToTime = (int) (Convert.ToInt32(toTime) *
          COEFFICIENT + ADDITION);

        // Create a new line using the new 'from' and 'to' times
        string fixedLine = "{" + newFromTime + "}" + "{" +
          newToTime + "}" + line.Substring(bracketToIndex + 1);

        return fixedLine;
    }
}
```

Again, we use the already familiar method for **reading a file line by line**. The difference this time is, that in the body of the loop, we write every line of the file with already corrected subtitles, after we have fixed them with the method **FixLine(string)** (this method is not the subject of our discussion, since it can be implemented in many different ways depending on what you want to adjust). Because we use the **using** block for both files, we can guarantee that they will be closed even if an exception occurs during processing (this may happen, for example if one of the lines in the file is not in the expected format).

Exercises

1. Write a program that reads a text file and **prints its odd lines** on the console.

2. Write a program that **joins two text files** and records the results in a third file.

3. Write a program that reads the contents of a text file and **inserts the line numbers** at the beginning of each line, then rewrites the file contents.

4. Write a program that **compares two text files** with the same number of rows line by line and prints the number of equal and the number of different lines.

5. Write a program that reads a square matrix of integers from a file and **finds the sub-matrix with size 2 × 2 that has the maximal sum** and writes this sum to a separate text file. The first line of input file contains the size of the recorded matrix (N). The next N lines contain N integers separated by spaces.

   Sample input file:

```
4
2 3 3 4
```

```
0 2 3 4
3 7 1 2
4 3 3 2
```

Sample output: 17.

6. Write a program that **reads a list of names** from a text file, arranges them in **alphabetical order**, and writes them to another file. The lines are written one per row.

7. Write a program that **replaces every occurrence of the substring** "**start**" with "**finish**" in a text file. Can you rewrite the program to replace whole words only? Does the program work for large files (e.g. 800 MB)?

8. Write the previous program so that it changes only the **whole words** (not parts of the word).

9. Write a program that **deletes all the odd lines** of a text file.

10. Write a program that extracts from an XML file the **text only** (without the tags). Sample input file:

```
<?xml version="1.0"><student><name>Peter</name>
<age>21</age><interests count="3"><interest>
Games</interest><interest>C#</interest><interest>
Java</interest></interests></student>
```

Sample output:

```
Peter
21
Games
C#
Java
```

11. Write a program that **deletes all words** that begin with the word "**test**". The words will contain only the following chars: 0…9, a…z, A…Z.

12. A text file **words.txt** is given, containing a list of words, one per line. Write a program that **deletes in the file text.txt all the words that occur in the other file**. Catch and handle all possible exceptions.

13. Write a program that **reads a list of words** from a file called **words.txt**, **counts how many times each of these words is found in another file text.txt**, and records the results in a third file – **result.txt**, but before that, sorts them by the number of occurrences in descending order. Handle all possible exceptions.

Solutions and Guidelines

1. Use the **examples** discussed in **this chapter**. Use the **using** structure to ensure proper closing of the input and the resulting stream.

2. You will have to first **read the input file line by line** and save it in the resulting file in **overwrite** mode. Then you must open the second input file and save it line by line in the result file in append mode. To create a **StreamWriter** in overwrite / use mode use the appropriate constructor (find it in MSDN).

   An alternative way is to read both files in a **string** with **ReadToEnd()**, store them in memory and save them in the resulting file. This approach does not work for large files (the likes of several gigabytes).

3. Follow the **examples in this chapter**. Think of how you would solve the task if the file were large (several gigabytes).

4. Follow the **examples in this chapter**. You will have to open both files simultaneously and read them line by line in a loop. If you reach the end of the (read null) file, that does not match the other's, that means that both files have different number of rows and an exception should be thrown.

5. Read the first line of the file and **create a matrix** with the specified size. After that read the following lines, line by line and separate the numbers. Then save them in the corresponding (row, column) in the matrix. Finally, find the sub-matrix using **two nested loops**.

6. Write each read name in a list (**List<string>**), then sort it properly (look for information on the method **Sort()**) and then print it in the result file.

7. Read the file **line by line** and use the methods of the class **String**. If you load the entire file into memory, instead of reading it line by line, problems when loading large files might occur.

8. For every occurrence of "start", check if that is the whole word or just a part of it.

9. Look at the examples in this chapter.

10. Read the input file **character by character**. When you encounter a "**<**", then this is a **starting tag**, but when you encounter a "**>**", that means a **closing tag**. All characters you encounter outside of the tags build up the text that must be extracted. You can store it in **StringBuilder** and print its contents when you encounter "**<**" or reach the end of the file.

11. Read the file **line by line** and **replace** words that start with "**test**" with an empty string. Use **Regex.Replace(…)** with an appropriate regular expression. Alternatively, you can search in the line the substring "**test**" and every time you find it, get all of its neighboring letters to the left and right. This way you find the word in which the string "**test**" is part of and you can delete it if it begins with "**test**".

12. The task is **similar to the previous one**. You can read the text **line by line** and replace each of the given words with an empty string. Test whether your program properly handles exceptions by simulating different scenarios (e.g. no file, lack of rights for reading and writing, etc.).

13. Create a **hash table with keys – the words from words.txt** and **value number of occurrences** of each word (**Dictionary<string, int>**). Firstly, save to the hash table that all words are found 0 times. Then read the file line by line and split each line into words. Check whether each obtained word can be found in the hash table, and if so, add 1 to the number of occurrences. Finally, save all the words and their number of occurrences in an array of type **KeyValuePair<string, int>**. Sort the array with a suitable comparison function like so:

```
Array.Sort<KeyValuePair<string, int>>(
  arr, (a, b) => a.Value.CompareTo(b.Value));
```

Chapter 16. Linear Data Structures

In This Chapter

In this chapter we are going to get familiar with some of the basic presentations of data in programming: **lists and linear data structures**. Very often in order to solve a given problem we need to work with a sequence of elements. For example, to read completely this book we have to read sequentially each page, i.e. to **traverse sequentially** each of the elements of the set of the pages in the book. Depending on the task, we have to apply different operations on this set of data. In this chapter we will introduce the concept of **abstract data types (ADT)** and will explain how a certain ADT can have **multiple different implementations**. After that we shall explore how and when to use **lists** and their implementations (**linked list**, **doubly-linked list** and **array-list**). We are going to see how for a given task one structure may be more convenient than another. We are going to consider the **structures "stack" and "queue"**, as well as their applications. We are going to get familiar with some **implementations** of these structures.

Abstract Data Structures

Before we start considering classes in C#, which implement some of the most frequently, used data structures (such as lists and queues), we are going to consider the concepts of **data structures** and **abstract data structures**.

What Is a Data Structure?

Very often, when we write programs, we have to work with many objects (data). Sometimes we add and remove elements, other times we would like to order them or to process the data in another specific way. For this reason, different ways of storing data are developed, depending on the task. Most frequently these elements are ordered in some way (for example, object A is before object B).

At this point we come to the aid of **data structures – a set of data** organized on the basis of logical and mathematical laws. Very often the choice of the right data structure makes the program much more efficient – we could save memory and execution time (and sometimes even the amount of code we write).

What Is an Abstract Data Type?

In general, **abstract data types (ADT)** gives us a definition (abstraction) of the specific structure, i.e. defines the allowed operations and properties, without being interested in the specific implementation. This allows an abstract data type to have several different implementations and respectively different efficiency.

Basic Data Structures in Programming

We can differentiate several groups of data structures:

- **Linear** – these include lists, stacks and queues
- **Tree-like** – different types of trees like binary trees, B-trees and balanced trees
- **Dictionaries** – key-value pairs organized in hash tables
- **Sets** – unordered bunches of unique elements
- **Others** – multi-sets, bags, multi-bags, priority queues, graphs, …

In this chapter we are going to explore the **linear (list-like) data structures**, and in the next several chapters we are going to pay attention to more complicated data structures, such as trees, graphs, hash tables and sets, and we are going to explain how and when to use each of them.

Mastering basic data structures in programming is essential, as without them we could not program efficiently. They, together with algorithms, are in the basis of programming and in the next several chapters we are going to get familiar with them.

List Data Structures

Most commonly used data structures are the **linear (list) data structures**. They are an abstraction of all kinds of rows, sequences, series and others from the real world.

List

We could imagine the **list** as an **ordered sequence (line) of elements**. Let's take as an example purchases from a shop. In the list we can read each of the elements (the purchases), as well as add new purchases in it. We can also remove (erase) purchases or shuffle them.

Abstract Data Structure "List"

Let's now give a more strict definition of the **structure list**:

List is a linear data structure, which contains a sequence of elements. The list has the property **length** (count of elements) and its elements are **arranged consecutively**.

The list allows adding elements on different positions, removing them and incremental crawling. Like we already mentioned, an ADT can have several implementations. An example of such ADT is the interface **System. Collections.IList**.

Interfaces in C# construct a frame (contract) for their implementations – classes. This contract consists of **a set of methods and properties**, which each class must implement in order to implement the interface. The data type "Interface" in C# we are going to discuss in depth in the chapter "Object-Oriented Programming Principles".

Each ADT defines some interface. Let's consider the interface **System. Collections.IList**. The basic methods, which it defines, are:

- **int Add(object)** – adds element in the end of the list
- **void Insert(int, object)** – adds element on a preliminary chosen position in the list
- **void Clear()** – removes all elements in the list
- **bool Contains(object)** – checks whether the list contains the element
- **void Remove(object)** – removes the element from the list
- **void RemoveAt(int)** – removes the element on a given position
- **int IndexOf(object)** – returns the position of the element
- **this[int]** – indexer, allows access to the elements on a set position

Let's see several from the basic implementations of the ADT list and explain in which situations they should be used.

Static List (Array-Based Implementation)

Arrays perform many of the features of the ADT list, but there is a significant difference – the lists allow adding new elements, while arrays have fixed size.

Despite of that, an implementation of list is possible with an array, which automatically increments its size (similar to the class **StringBuilder**, which we already know from the chapter "Strings"). Such list is called **static list implemented with an extensible array**. Below we shall give a sample implementation of auto-resizable array-based list (array list). It is intended to hold any data type **T** through the concept of **generics** (see the "Generics" section in chapter "Defining Classes"):

```
public class CustomArrayList<T>
{
    private T[] arr;
    private int count;

    /// <summary>Returns the actual list length</summary>
```

```
public int Count
{
    get
    {
        return this.count;
    }
}

private const int INITIAL_CAPACITY = 4;

/// <summary>
/// Initializes the array-based list – allocate memory
/// </summary>
public CustomArrayList(int capacity = INITIAL_CAPACITY)
{
    this.arr = new T[capacity];
    this.count = 0;
}
```

Firstly, we define an **array**, in which we are going **to keep the elements**, as well as a counter for the current count of elements. After that we add the constructor, as we initialize our array with some **initial capacity** (when capacity is not specified) in order to avoid resizing it when adding the first few elements. Let's take a look at some typical operations like **add** (append) an element, **insert** an element at specified position (index) and **clear** the list:

```
/// <summary>Adds element to the list</summary>
/// <param name="item">The element you want to add</param>
public void Add(T item)
{
    GrowIfArrIsFull();
    this.arr[this.count] = item;
    this.count++;
}

/// <summary>
/// Inserts the specified element at given position in this list
/// </summary>
/// <param name="index">
/// Index, at which the specified element is to be inserted
/// </param>
/// <param name="item">Element to be inserted</param>
/// <exception cref="System.IndexOutOfRangeException">Index is
invalid</exception>
public void Insert(int index, T item)
```

```
{
    if (index > this.count || index < 0)
    {
        throw new IndexOutOfRangeException(
            "Invalid index: " + index);
    }
    GrowIfArrIsFull();
    Array.Copy(this.arr, index,
        this.arr, index + 1, this.count - index);
    this.arr[index] = item;
    this.count++;
}

/// <summary>
/// Doubles the size of this.arr (grow) if it is full
/// </summary>
private void GrowIfArrIsFull()
{
    if (this.count + 1 > this.arr.Length)
    {
        T[] extendedArr = new T[this.arr.Length * 2];
        Array.Copy(this.arr, extendedArr, this.count);
        this.arr = extendedArr;
    }
}

/// <summary>Clears the list (remove everything)</summary>
public void Clear()
{
    this.arr = new T[INITIAL_CAPACITY];
    this.count = 0;
}
```

We implemented the operation **adding** a new element, as well as **inserting** a new element which both first ensure that the internal array (buffer) holding the elements has enough capacity. If the internal buffer is full, it is extended (grown) to a double of the current capacity. Since arrays in .NET do not support resizing, the **growing operation** allocated a new array of double size and moves all elements from the old array to the new.

Below we implement **searching** operations (finding the index of given element and checking whether given element exists), as well as **indexer** – the ability to access the elements (for read and change) by their index specified in the **[]** operator:

```
/// <summary>
/// Returns the index of the first occurrence of the specified
/// element in this list (or -1 if it does not exist).
/// </summary>
/// <param name="item">The element you are searching</param>
/// <returns>
/// The index of a given element or -1 if it is not found
/// </returns>
public int IndexOf(T item)
{
   for (int i = 0; i < this.arr.Length; i++)
   {
      if (object.Equals(item, this.arr[i]))
      {
         return i;
      }
   }

   return -1;
}

/// <summary>Checks if an element exists</summary>
/// <param name="item">The item to be checked</param>
/// <returns>If the item exists</returns>
public bool Contains(T item)
{
   int index = IndexOf(item);
   bool found = (index != -1);
   return found;
}

/// <summary>Indexer: access to element at given index</summary>
/// <param name="index">Index of the element</param>
/// <returns>The element at the specified position</returns>
public T this[int index]
{
   get
   {
      if (index >= this.count || index < 0)
      {
         throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
      }
      return this.arr[index];
```

```
   }
   set
   {
      if (index >= this.count || index < 0)
      {
         throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
      }
      this.arr[index] = value;
   }
}
```

We add operations for **removing** items (by index and by value):

```
/// <summary>Removes the element at the specified index
/// </summary>
/// <param name="index">The index of the element to remove
/// </param>
/// <returns>The removed element</returns>
public T RemoveAt(int index)
{
   if (index >= this.count || index < 0)
   {
      throw new ArgumentOutOfRangeException(
         "Invalid index: " + index);
   }

   T item = this.arr[index];
   Array.Copy(this.arr, index + 1,
      this.arr, index, this.count - index - 1);
   this.arr[this.count - 1] = default(T);
   this.count--;

   return item;
}

/// <summary>Removes the specified item</summary>
/// <param name="item">The item to be removed</param>
/// <returns>The removed item's index or -1 if the item does not
exist</returns>
public int Remove(T item)
{
   int index = IndexOf(item);
   if (index != -1)
```

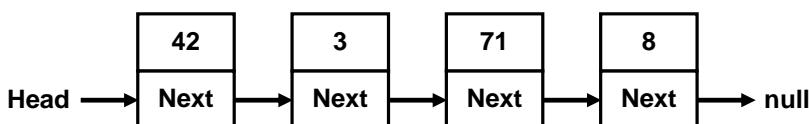
```
    {
        this.RemoveAt(index);
    }
    return index;
}
```

In the methods above we **remove** elements. For this purpose, firstly we find the searched element, remove it and then shift the elements after it by one position to the left, in order to fill the empty position. Finally, we fill the position after the last item in the array with **null** value (the **default(T)**) to allow the garbage collector to release it if it is not needed. Generally, we want to keep all unused elements in the **arr** empty (**null** / zero value).

Let's consider a sample usage of the recently implemented class. There is a **Main()** method, in which we demonstrate most of the operations. In the enclosed code we create a list of purchases, add, insert and remove few items and print the list on the console. Finally we check whether certain items exist:

```
class CustomArrayListTest
{
    static void Main()
    {
        CustomArrayList<string> shoppingList =
            new CustomArrayList<string>();
        shoppingList.Add("Milk");
        shoppingList.Add("Honey");
        shoppingList.Add("Olives");
        shoppingList.Add("Water");
        shoppingList.Add("Beer");
        shoppingList.Remove("Olives");
        shoppingList.Insert(1, "Fruits");
        shoppingList.Insert(0, "Cheese");
        shoppingList.Insert(6, "Vegetables");
        shoppingList.RemoveAt(0);
        shoppingList[3] = "A lot of " + shoppingList[3];
        Console.WriteLine("We need to buy:");
        for (int i = 0; i < shoppingList.Count; i++)
        {
            Console.WriteLine(" - " + shoppingList[i]);
        }
        Console.WriteLine("Position of 'Beer' = {0}",
            shoppingList.IndexOf("Beer"));
        Console.WriteLine("Position of 'Water' = {0}",
            shoppingList.IndexOf("Water"));
        Console.WriteLine("Do we have to buy Bread? " +
```

```
        shoppingList.Contains("Bread"));
    }
}
```

Here is how the output of the program execution looks like:

```
We need to buy:
 - Milk
 - Fruits
 - Honey
 - A lot of Water
 - Beer
 - Vegetables
Position of 'Beer' = 4
Position of 'Water' = -1
Do we have to buy Bread? False
```

Linked List (Dynamic Implementation)

As we saw, the static list has a serious disadvantage – the operations for inserting and removing items from the inside of the array requires rearrangement of the elements. When frequently inserting and removing items (especially a large number of items), this can lead to low performance. In such cases it is advisable to use the so called **linked lists**. The difference in them is the structure of elements – while in the static list the element contains only the specific object, with the dynamic list the **elements keep information about their next element**.

Here is how a **sample linked list** looks like in the memory:

For the dynamic **implementation of the linked list** we will need two classes: the class **ListNode**, which will hold a single element of the list along with its next element, and the main list class **DynamicList<T>** which will hold a sequence of elements as well as the **head** and the **tail** of the list:

```
/// <summary>Dynamic (linked) list class definition</summary>
public class DynamicList<T>
{
    private class ListNode
    {
        public T Element { get; set; }
        public ListNode NextNode { get; set; }
```

```
        public ListNode(T element)
        {
            this.Element = element;
            NextNode = null;
        }

        public ListNode(T element, ListNode prevNode)
        {
            this.Element = element;
            prevNode.NextNode = this;
        }
    }

    private ListNode head;
    private ListNode tail;
    private int count;

    // …
}
```

First, let's consider the recursive class **ListNode**. It holds a single element and a reference (pointer) to the next element which is of the same class **ListNode**. So **ListNode** is an example of **recursive data structure** that is defined by referencing itself. The class is inner to the class **DynamicList<T>** (it is declared as a private member) and is therefore accessible only to it. For our **DynamicList<T>** we create 3 fields: **head** – pointer to the first element, **tail** – pointer to the last element and **count** – counter of the elements.

After that we declare the **constructor** which creates and empty linked list:

```
public DynamicList()
{
    this.head = null;
    this.tail = null;
    this.count = 0;
}
```

Upon the initial construction the list is empty and for this reason we assign **head** = **tail** = **null** and **count** = **0**.

We are going to implement all basic operations: **adding** and **removing** items, as well as **searching** for an element and **accessing the elements by index**.

Let's start with the operation **add** (append) which is relatively simple. Two cases are considered: an **empty list** and a **non-empty list**. In both cases we

append the element at the end of the list (where **tail** points) and after the operation all fields (**head**, **tail** and **count**) have correct values:

```
/// <summary>Add element at the end of the list</summary>
/// <param name="item">The element to be added</param>
public void Add(T item)
{
   if (this.head == null)
   {
      // We have an empty list -> create a new head and tail
      this.head = new ListNode(item);
      this.tail = this.head;
   }
   else
   {
      // We have a non-empty list -> append the item after tail
      ListNode newNode = new ListNode(item, this.tail);
      this.tail = newNode;
   }
   this.count++;
}
```

You can now see the operation **removing** an item at specified index. It is considerably more complicated than adding:

```
/// <summary>Removes and returns element on the specified index
/// </summary>
/// <param name="index">The index of the element to be removed
/// </param>
/// <returns>The removed element</returns>
/// <exception cref="System.ArgumentOutOfRangeException">
/// if the index is invalid</exception>
public T RemoveAt(int index)
{
   if (index >= count || index < 0)
   {
      throw new ArgumentOutOfRangeException(
         "Invalid index: " + index);
   }

   // Find the element at the specified index
   int currentIndex = 0;
   ListNode currentNode = this.head;
   ListNode prevNode = null;
   while (currentIndex < index)
```

```
    {
        prevNode = currentNode;
        currentNode = currentNode.NextNode;
        currentIndex++;
    }

    // Remove the found element from the list of nodes
    RemoveListNode(currentNode, prevNode);

    // Return the removed element
    return currentNode.Element;
}

/// <summary>
/// Remove the specified node from the list of nodes
/// </summary>
/// <param name="node">the node for removal</param>
/// <param name="prevNode">the predecessor of node</param>
private void RemoveListNode(ListNode node, ListNode prevNode)
{
    count--;
    if (count == 0)
    {
        // The list becomes empty -> remove head and tail
        this.head = null;
        this.tail = null;
    }
    else if (prevNode == null)
    {
        // The head node was removed --> update the head
        this.head = node.NextNode;
    }
    else
    {
        // Redirect the pointers to skip the removed node
        prevNode.NextNode = node.NextNode;
    }

    // Fix the tail in case it was removed
    if (object.ReferenceEquals(this.tail, node))
    {
        this.tail = prevNode;
    }
}
```

Firstly, we **check if the specified index exists**, and if it does not, an appropriate exception is thrown. After that, the element for removal is found by moving forward from the beginning of the list to the next element **exactly index times**. After the element for removal has been found (**currentNode**), it is removed by the additional private method **RemoveListNode(…)**, which considers the following 3 possible cases:

- The **list remains empty after the removal** → we remove the whole list along with its head and tail (**head = null**, **tail = null**, **count = 0**).
- The element for removal is **at the start of the list** (there is no previous element) → we make **head** to point at the element immediately after the removed element (or at **null**, if the removed element was the last one).
- The element is **in the middle or at the end of the list** → we direct the element before it to point to the element after it (or at **null**, if there is no next element).

Finally, we make sure **tail points to the end of the list** (in case **tail** was pointed to the removed element, it is fixed to point to its predecessor).

The next is the implementation of the **removal** of an element by its value:

```
/// <summary>
/// Removes the specified item and return its index.
/// </summary>
/// <param name="item">The item for removal</param>
/// <returns>The index of the element or -1 if it does not
exist</returns>
public int Remove(T item)
{
   // Find the element containing the searched item
   int currentIndex = 0;
   ListNode currentNode = this.head;
   ListNode prevNode = null;
   while (currentNode != null)
   {
      if (object.Equals(currentNode.Element, item))
      {
         break;
      }
      prevNode = currentNode;
      currentNode = currentNode.NextNode;
      currentIndex++;
   }

   if (currentNode != null)
   {
```

```
        // The element is found in the list -> remove it
        RemoveListNode(currentNode, prevNode);
        return currentIndex;
    }
    else
    {
        // The element is not found in the list -> return -1
        return -1;
    }
}
```

The removal by value of an element works **like the removal of an element by index**, but there are two special considerations: the searched element **may not exist** and for this reason an extra check is necessary; there may be elements with **null** value in the list, which have to be removed and processed correctly. The last is done by comparing the elements through the static method **object.Equals(…)** which works well with **null** values.

In order the removal to work correctly, it is necessary the elements in the array to be comparable, i.e. to have a correct implementation of the method **Equals()** derived from **System.Object**.

Bellow we give implementations of the operations for **searching** and checking whether the list **contains** a specified element:

```
/// <summary>Searches for given element in the list</summary>
/// <param name="item">The item to be searched</param>
/// <returns>
/// The index of the first occurrence of the element
/// in the list or -1 when it is not found
/// </returns>
public int IndexOf(T item)
{
    int index = 0;
    ListNode currentNode = this.head;
    while (currentNode != null)
    {
        if (object.Equals(currentNode.Element, item))
        {
            return index;
        }
        currentNode = currentNode.NextNode;
        index++;
    }
    return -1;
}
```

```
/// <summary>
/// Checks if the specified element exists in the list
/// </summary>
/// <param name="item">The item to be checked</param>
/// <returns>
/// True if the element exists or false otherwise
/// </returns>
public bool Contains(T item)
{
   int index = IndexOf(item);
   bool found = (index != -1);
   return found;
}
```

The searching for an element works **like in the method for removing**: we start from the beginning of the list and check sequentially the next elements one after another, until we reach the end of the list or find the searched element.

We have two more operations to implement – **accessing elements by index** (using the indexer) and finding the **count of elements** (through a property):

```
/// <summary>
/// Gets or sets the element at the specified position
/// </summary>
/// <param name="index">
/// The position of the element [0 … count-1]
/// </param>
/// <returns>The item at the specified index</returns>
/// <exception cref="System.ArgumentOutOfRangeException">
/// When an invalid index is specified
/// </exception>
public T this[int index]
{
   get
   {
      if (index >= count || index < 0)
      {
         throw new ArgumentOutOfRangeException(
            "Invalid index: " + index);
      }
      ListNode currentNode = this.head;
      for (int i = 0; i < index; i++)
      {
```

```
            currentNode = currentNode.NextNode;
        }
        return currentNode.Element;
    }
    set
    {
        if (index >= count || index < 0)
        {
            throw new ArgumentOutOfRangeException(
                "Invalid index: " + index);
        }
        ListNode currentNode = this.head;
        for (int i = 0; i < index; i++)
        {
            currentNode = currentNode.NextNode;
        }
        currentNode.Element = value;
    }
}

/// <summary>
/// Gets the count of elements in the list
/// </summary>
public int Count
{
    get
    {
        return this.count;
    }
}
```

The indexer works pretty straightforward – first checks the validity of the specified index and then starts from the **head** of the list goes to the next node **index** times. Once the node containing the element the specified **index** is found, it is accessed directly.

Let's finally see a **shopping list example** similar to the example with the static list implementation, this time using with our **linked list**:

```
class DynamicListTest
{
    static void Main()
    {
        DynamicList<string> shoppingList =
            new DynamicList<string>();
```

```
        shoppingList.Add("Milk");
        shoppingList.Remove("Milk"); // Empty list
        shoppingList.Add("Honey");
        shoppingList.Add("Olives");
        shoppingList.Add("Water");
        shoppingList[2] = "A lot of " + shoppingList[2];
        shoppingList.Add("Fruits");
        shoppingList.RemoveAt(0); // Removes "Honey" (first)
        shoppingList.RemoveAt(2); // Removes "Fruits" (last)
        shoppingList.Add(null);
        shoppingList.Add("Beer");
        shoppingList.Remove(null);
        Console.WriteLine("We need to buy:");
        for (int i = 0; i < shoppingList.Count; i++)
        {
            Console.WriteLine(" - " + shoppingList[i]);
        }
        Console.WriteLine("Position of 'Beer' = {0}",
            shoppingList.IndexOf("Beer"));
        Console.WriteLine("Position of 'Water' = {0}",
            shoppingList.IndexOf("Water"));
        Console.WriteLine("Do we have to buy Bread? " +
            shoppingList.Contains("Bread"));
    }
}
```

The above code **checks all the operations** from our linked list implementation along with their special cases (like removing the first and the last element) and shows that out dynamic list implementation works correctly. The output of the above code is the following:

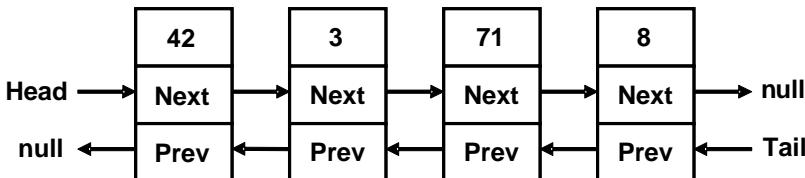
```
We need to buy:
 - Olives
 - A lot of Water
 - Beer
Position of 'Beer' = 2
Position of 'Water' = -1
Do we have to buy Bread? False
```

Comparing the Static and the Dynamic Lists

We implemented the abstract data type (ADT) **list** in two ways: **static (array list)** and **dynamic (linked list)**. Once written these two implementations can be used in almost exactly the same way. For example see the following two pieces of code (using our array list and our linked list):

```
static void Main()
{
  CustomArrayList<string> arrayList =
    new CustomArrayList<string>();
  arrayList.Add("One");
  arrayList.Add("Two");
  arrayList.Add("Three");
  arrayList[0] = "Zero";
  arrayList.RemoveAt(1);
  Console.WriteLine("Array list: ");
  for (int i = 0; i < arrayList.Count; i++)
  {
    Console.WriteLine(" - " + arrayList[i]);
  }

  DynamicList<string> dynamicList =
    new DynamicList<string>();
  dynamicList.Add("One");
  dynamicList.Add("Two");
  dynamicList.Add("Three");
  dynamicList[0] = "Zero";
  dynamicList.RemoveAt(1);
  Console.WriteLine("Dynamic list: ");
  for (int i = 0; i < dynamicList.Count; i++)
  {
    Console.WriteLine(" - " + dynamicList[i]);
  }
}
```

The result of using the two types of lists is the same:

```
Array list:
 - Zero
 - Three
Dynamic list:
 - Zero
 - Three
```

The above example demonstrates that certain ADT could be implemented in several conceptually different ways and the users may not notice the difference between them. Still, different implementations could have different performance and could take different amount of memory.

This concept, known as **abstract behavior**, is fundamental for OOP and can be implemented by **abstract classes** or **interfaces** as we shall see in the section "Abstraction" of chapter "Object-Oriented Programming Principles".

Doubly-Linked List

In the so called **doubly-linked lists** each element contains its **value** and **two pointers – to the previous and to the next element** (or `null`, if there is no such element). This allows us to traverse the list forward and backward and some operations to be implemented more efficiently. Here is how a sample doubly-linked list looks like:

The ArrayList Class

After we got familiar with some of the basic implementations of the lists, we are going to consider the classes in C#, which deliver list data structures "without lifting a finger". The first one is the class `ArrayList`, which is an **untyped dynamically-extendable array**. It is implemented similarly to the static list implementation, which we considered earlier. `ArrayList` gives the opportunity to add, delete and search for elements in it. Some more important class members we may use are:

- **`Add(object)`** – adding a new element
- **`Insert(int, object)`** – adding a new element at a specified position (**`index`**)
- **`Count`** – returns the count of elements in the list
- **`Remove(object)`** – removes a specified element
- **`RemoveAt(int)`** – removes the element at a specified position
- **`Clear()`** – removes all elements from the list
- **`this[int]`** – an indexer, allows accessing the elements by a given position (index)

As we saw, one of the main problems with this implementation is the resizing of the inner array when adding and removing elements. In the `ArrayList` the problem is solved by preliminarily created array (buffer), which gives us the opportunity to add elements without resizing the array at each insertion or removal of elements.

The ArrayList Class – Example

The `ArrayList` class is **untyped**, so it can keep all kinds of elements – numbers, strings and other objects. Here is a small example:

```
using System;
using System.Collections;

class ProgrArrayListExample
{
  static void Main()
  {
    ArrayList list = new ArrayList();
    list.Add("Hello");
    list.Add(5);
    list.Add(3.14159);
    list.Add(DateTime.Now);

    for (int i = 0; i < list.Count; i++)
    {
      object value = list[i];
      Console.WriteLine("Index={0}; Value={1}", i, value);
    }
  }
}
```

In the example we create **ArrayList** and we add in it several elements from different types: **string**, **int**, **double** and **DateTime**. After that we iterate over the elements and print them. If we execute the example, we are going to get the following result:

```
Index=0; Value=Hello
Index=1; Value=5
Index=2; Value=3.14159
Index=3; Value=29.12.2009 23:17:01
```

ArrayList of Numbers – Example

In case we would like to make an array of numbers and then process them, for example to find their sum, we have to convert the **object** type to a number. This is because **ArrayList** is actually a list of elements of type **object**, and not from some specific type (like **int** or **string**). Here is a sample code, which sums the elements of **ArrayList**:

```
ArrayList list = new ArrayList();
list.Add(2);
list.Add(3.5f);
list.Add(25u);
list.Add(" EUR");
dynamic sum = 0;
```

```
for (int i = 0; i < list.Count; i++)
{
  dynamic value = list[i];
  sum = sum + value;
}
Console.WriteLine("Sum = " + sum);
// Output: Sum = 30.5 EUR
```

Note that in the array list we hold different types of values (**int**, **float**, **uint** and **string**) and we sum them in a variable of special type called **dynamic**. In C# **dynamic** is a universal data type intended to hold any value (numbers, objects, strings, even functions and methods). Operations over **dynamic** variables (like the **+** operator used above) are **resolved at runtime** and their action depends on the actuals values of their arguments. At compile time almost every operation with **dynamic** variables successfully compiles. At runtime, if the operation can be performed, it is performed, otherwise and exception is thrown. This explains why we apply the operation **+** over the arguments **2**, **3.5f**, **25u** and **" EUR"** and we finally obtain as a result the string **"30.5 EUR"**.

Generic Collections

Before we continue to play with more examples of working with the **ArrayList** class, we shall recall the concept of Generic Data Types in C#, which gives the opportunity to parameterize lists and collections in C#.

When we use the **ArrayList** class and all classes, which implement the interface **System.IList**, we face the problem we saw earlier: when we add a new element from a class, we pass it as a value of type **object**. Later, when we search for a certain element, we get it as **object** and we have to cast it to the expected type (or use **dynamic**). It is not guaranteed, however, that all elements in the list will be of one and the same type. Besides this, the conversion from one type to another takes time, and this drastically slows down the program execution.

To solve the problem we use the **generic (template / parameterized) classes**. They are created to work with one or several types, as when we create them, we indicate what type of objects we are going to keep in them. Let's recall that we create an instance of a generic class, for example **GenericType**, by indicating the type, of which the elements have to be:

```
GenericType<T> instance = new GenericType<T>();
```

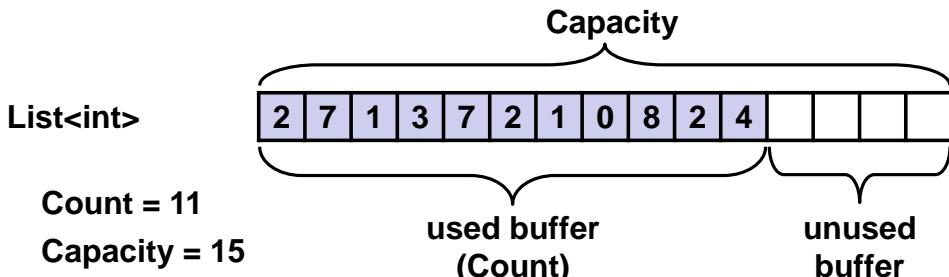
This type **T** can be any successor of the class **System.Object**, for example **string** or **DateTime**. Here are few examples:

```
List<int> intList = new List<int>();
```

```
List<bool> boolList = new List<bool>();
List<double> realNumbersList = new List<double>();
```

Let's consider some of the **generic collections in .NET Framework**.

The List<T> Class

`List<T>` is the generic variant of `ArrayList`. When we create an object of type **`List<T>`**, we indicate the type of the elements, which will be hold in the list, i.e. we substitute the denoted by **`T`** type with some real data type (for example number or string).

Let's consider a case in which we would like to create a list of integer elements. We could do this in the following way:

```
List<int> intList = new List<int>();
```

Thus the created list can contain only integer numbers and cannot contain other objects, for example strings. If we try to add to **`List<int>`** an object of type **`string`**, we are going to get a compilation error. Via the generic types the C# compiler protects us from mistakes when working with collections.

The List Class – Array-Based Implementation

`List<T>` works like our class **`CustomArrayList<T>`**. It keeps its elements in the memory as an **array**, which is **partially in use and partially free** for new elements (blank). Thanks to the reserved blank elements in the array, the operation **append** almost always manages to add the new element without the need to resize the array. Sometimes, of course, the array has to be resized, but as each resize would double the size of the array, resizing happens so seldom that it can be ignored in comparison to the count of append operations. We could imagine a **`List<T>`** like an **array, which has some capacity and is filled to a certain level**:

Thanks to the preliminarily allocated capacity of the array, containing the elements of the class **`List<T>`**, it can be extremely efficient data structure when it is necessary to **add elements fast**, **extract elements** and **access the elements by index**. Still, it is pretty **slow in inserting and removing elements** unless these elements are at the last position.

We could say that **List<T>** combines the good sides of lists and arrays – fast adding, changeable size and direct access by index.

When to Use List<T>?

We already explained that the **List<T>** class uses an inner array for keeping the elements and the array doubles its size when it gets overfilled. Such implementation causes the following good and bad sides:

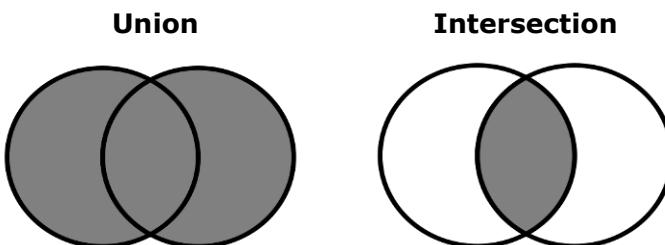
- The **search by index is very fast** – we can access with equal speed each of the elements, regardless of the count of elements.
- The **search for an element by value** works with as many comparisons as the count of elements (in the worst case), i.e. it **is slow**.
- **Inserting and removing** elements is a **slow** operation – when we add or remove elements, especially if they are not in the end of the array, we have to shift the rest of the elements and this is a slow operation.
- When **adding a new element**, sometimes we have to increase the capacity of the array, which is a slow operation, but it happens seldom and the average speed of insertion to **List** does not depend on the count of elements, i.e. it works **very fast**.

Use List<T> when you don't expect frequent insertion and deletion of elements, but you expect to add new elements at the end of the list or to access the elements by index.

Prime Numbers in Given Interval – Example

After we got familiar with the implementation of the structure list and the class **List<T>**, let's see how to use them. We are going to consider the problem for **finding the prime numbers in a certain interval**. For this purpose we have to use the following algorithm:

```
static List<int> GetPrimes(int start, int end)
{
  List<int> primesList = new List<int>();
  for (int num = start; num <= end; num++)
  {
    bool prime = true;
    double numSqrt = Math.Sqrt(num);
    for (int div = 2; div <= numSqrt; div++)
    {
      if (num % div == 0)
      {
        prime = false;
        break;
      }
```

```
        }
        if (prime)
        {
            primesList.Add(num);
        }
    }
    return primesList;
}

static void Main()
{
    List<int> primes = GetPrimes(200, 300);
    foreach (var item in primes)
    {
        Console.Write ("{0} ", item);
    }
}
```

From the mathematics we know that if a number is **not prime** it has **at least one divisor** in the interval [2 … square root from the given number]. This is what we use in the example above. For each number we look for a divisor in this interval. If we find a divisor, then the number is not prime and we could continue with the next number from the interval. Gradually, by adding the prime numbers we fill the list, after which we traverse it and print it on the screen. Here is how the output of the code above looks like:

```
211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293
```

Union and Intersection of Lists – Example

Let's consider a more interesting example – let's write a program, which can find the **union and the intersection of two sets of numbers**.

We could consider that there are two lists of numbers and we would like to take the elements, which are in both sets (**intersection**), or we look for those elements, which are at least in one of the sets (**union**).

Let's discuss one possible solution to the problem:

```
static List<int> Union(
    List<int> firstList, List<int> secondList)
{
  List<int> union = new List<int>();
  union.AddRange(firstList);
  foreach (var item in secondList)
  {
    if (!union.Contains(item))
    {
      union.Add(item);
    }
  }
  return union;
}

static List<int> Intersect(List<int>
    firstList, List<int> secondList)
{
  List<int> intersect = new List<int>();
  foreach (var item in firstList)
  {
    if (secondList.Contains(item))
    {
      intersect.Add(item);
    }
  }

  return intersect;
}

static void PrintList(List<int> list)
{
  Console.Write("{ ");
  foreach (var item in list)
  {
    Console.Write(item);
    Console.Write(" ");
  }
  Console.WriteLine("}");
}

static void Main()
{
  List<int> firstList = new List<int>();
```

```
    firstList.Add(1);
    firstList.Add(2);
    firstList.Add(3);
    firstList.Add(4);
    firstList.Add(5);
    Console.Write("firstList = ");
    PrintList(firstList);

    List<int> secondList = new List<int>();
    secondList.Add(2);
    secondList.Add(4);
    secondList.Add(6);
    Console.Write("secondList = ");
    PrintList(secondList);

    List<int> unionList = Union(firstList, secondList);
    Console.Write("union = ");
    PrintList(unionList);

    List<int> intersectList =
        Intersect(firstList, secondList);
    Console.Write("intersect = ");
    PrintList(intersectList);
}
```

The program logic in this solution **directly follows the definitions of union and intersection** of sets. We use the operations for searching for an element in a list and insertion of a new element in a list.

We are going to solve the problem in one more way: by using the method **AddRange<T>(IEnumerable<T> collection)** from the class **List<T>**:

```
static void Main()
{
    List<int> firstList = new List<int>();
    firstList.Add(1);
    firstList.Add(2);
    firstList.Add(3);
    firstList.Add(4);
    firstList.Add(5);
    Console.Write("firstList = ");
    PrintList(firstList);

    List<int> secondList = new List<int>();
    secondList.Add(2);
```

```
    secondList.Add(4);
    secondList.Add(6);
    Console.Write("secondList = ");
    PrintList(secondList);

    List<int> unionList = new List<int>();
    unionList.AddRange(firstList);
    for (int i = unionList.Count-1; i >= 0; i--)
    {
        if (secondList.Contains(unionList[i]))
        {
            unionList.RemoveAt(i);
        }
    }
    unionList.AddRange(secondList);
    Console.Write("union = ");
    PrintList(unionList);

    List<int> intersectList = new List<int>();
    intersectList.AddRange(firstList);
    for (int i = intersectList.Count-1; i >= 0; i--)
    {
        if (!secondList.Contains(intersectList[i]))
        {
            intersectList.RemoveAt(i);
        }
    }
    Console.Write("intersect = ");
    PrintList(intersectList);
}
```

In order to intersect the sets we do the following: we put all elements from the first list (via **AddRange()**), after which we remove all elements, which are not in the second list.

The problem can also be solved even in an easier way by using the method **RemoveAll(Predicate<T> match)**, but it is related to using programming constructs, called **delegates and lambda expressions**, which are considered in the chapter Lambda Expressions and LINQ. The union we make as we add elements from the first list, after which we remove all elements, which are in the second list, and finally we add all elements of the second list.

The result from the two programs **is exactly the same**:

```
firstList = { 1 2 3 4 5 }
secondList = { 2 4 6 }
```

```
union = { 1 2 3 4 5 6 }
intersect = { 2 4 }
```

Converting a List to Array and Vice Versa

In C# the **conversion of a list to an array** is easy by using the given method **ToArray**(). For the opposite operation we could use the constructor of **List<T>(System.Array).** Let's see an example, demonstrating their usage:

```
static void Main()
{
    int[] arr = new int[] { 1, 2, 3 };
    List<int> list = new List<int>(arr);
    int[] convertedArray = list.ToArray();
}
```

The LinkedList<T> Class

This class is a **dynamic implementation of a doubly linked list** built in .NET Framework. Its elements contain a certain value and a pointer to the previous and the next element. The **LinkedList<T>** class in .NET works in similar fashion like our class **DynamicList<T>**.

When Should We Use LinkedList<T>?

We saw that the dynamic and the static implementation have their specifics considering the different operations. With a view to the structure of the linked list, we have to have the following in mind:

- The **append** operation is **very fast**, because the list always knows its last element (**tail**).
- **Inserting** a new element at a random position in the list is **very fast** (unlike **List<T>**) if we have a pointer to this position, e.g. if we insert at the list start or at the list end.
- **Searching** for elements **by index** or **by value** in **LinkedList** is a **slow** operation, as we have to scan all elements consecutively by beginning from the start of the list.
- **Removing** elements is a **slow** operation, because it includes searching.

Basic Operations in the LinkedList<T> Class

LinkedList<T> has the same operations as in **List<T>**, which makes the two classes interchangeable, but in fact **List<T>** is used more often. Later we are going to see that **LinkedList<T>** is used when working with queues.

When Should We Use LinkedList<T>?

Using **LinkedList<T>** is preferable when we have to **add / remove elements at both ends of the list** and when the access to the elements is consequential.

However, when we have to **access the elements by index**, then **List<T>** is a more appropriate choice.

Considering memory, **LinkedList<T>** generally takes more space because it holds the value and several additional pointers for each element. **List<T>** also takes additional space because it allocates memory for more elements than it actually uses (it keeps bigger capacity than the number of its elements).

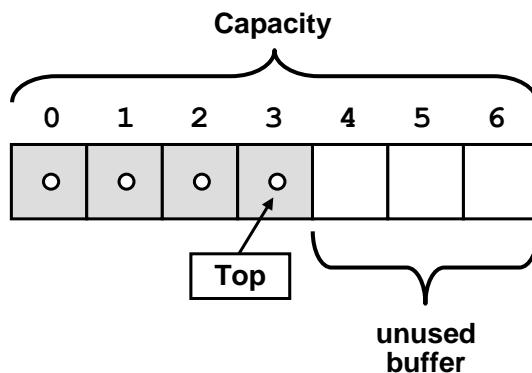
Stack

Let's imagine several cubes, which we have put one above other. We could put a new cube on the top, as well as remove the highest cube. Or let's imagine a chest. In order to take out the clothes on the bottom, first we have to take out the clothes above them.

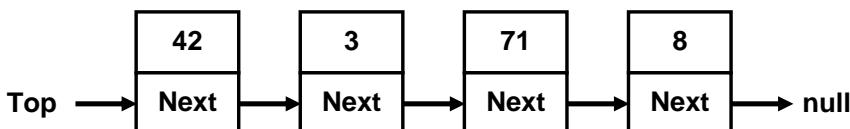
This is the classical data structure **stack** – we could add elements on the top and remove the element, which has been added last, but no the previous ones (the ones that are below it). In programming the **stack is a commonly used data structure**. The stack is used internally by the .NET virtual machine (CLR) for keeping the variables of the program and the parameters of the called methods (it is called **program execution stack**).

The Abstract Data Type "Stack"

The **stack** is a data structure, which implements the behavior "**last in – first out**" (**LIFO**). As we saw with the cubes, the elements could be added and removed only on the top of the stack.

ADT stack provides 3 major operations: **push** (add an element at the top of the stack), **pop** (take the last added element from the top of the stack) and **peek** (get the element form the top of the stack without removing it).

The data structure stack can also have different implementations, but we are going to consider two – **dynamic** and **static implementation**.

Static Stack (Array-Based Implementation)

Like with the static list we can use an **array to keep the elements** of the stack. We can keep an index or a pointer to the element, which is at the top.

Usually, if the internal array is filled, we have to resize it (to allocate twice more memory), like this happens with the static list (**ArrayList**, **List<T>** and **CustomArrayList<T>**). Unused buffer memory should be hold to ensure fast push and pop operations.

Here is how we could imagine a **static stack**:

Linked Stack (Dynamic Implementation)

For the **dynamic implementation of stack** we use elements, which keep a value and a pointer to the next element. This linked-list based implementation does not require an internal buffer, does not need to grow when the buffer is full and has virtually the same performance for the major operations like the static implementation:

When the stack is empty, the **top** has value **null**. When a new item is added, it is inserted on a position where the **top** indicates, after which the top is redirected to the new element. Removal is done by deleting the first element, pointed by **the** top pointer.

The Stack<T> Class

In C# we could use the standard implementation of the class in .NET Framework **System.Collections.Generics.Stack<T>**. It is **implemented statically with an array**, as the array is resized when needed.

The Stack<T> Class – Basic Operations

All basic operations for working with a stack are implemented:

- **Push(T)** – adds a new element on the top of the stack
- **Pop()** – returns the highest element and removes it from the stack
- **Peek()** – returns the highest element without removing it
- **Count** – returns the count of elements in the stack
- **Clear()** – retrieves all elements from the stack
- **Contains(T)** – check whether the stack contains the element
- **ToArray()** – returns an array, containing all elements of the stack

Stack Usage – Example

Let's take a look at a simple example on how to use stack. We are going to add several elements, after which we are going to print them on the console.

```
static void Main()
{
   Stack<string> stack = new Stack<string>();
   stack.Push("1. John");
   stack.Push("2. Nicolas");
   stack.Push("3. Mary");
   stack.Push("4. George");
   Console.WriteLine("Top = " + stack.Peek());
   while (stack.Count > 0)
   {
      string personName = stack.Pop();
      Console.WriteLine(personName);
   }
}
```

As the stack is a "**last in, first out**" (**LIFO**) structure, the program is going to print the records in a reversed order. Here is its output:

```
Top = 4. George
4. George
3. Mary
2. Nicolas
1. John
```

Correct Brackets Check – Example

Let's consider the following task: we have an expression, in which we would like to **check whether the brackets are put correctly**. This means to check if the count of the opening brackets is equal to the count of the closing brackets and all opening brackets match their respective closing brackets. The specification of the **stack** allows us to check whether the bracket we have met has a corresponding closing bracket. When we meet an opening bracket, we add it to the stack. When we meet a closing bracket, we remove an element from the stack. If the stack becomes empty before the end of the program in a moment when we have to remove an element, the brackets are incorrectly placed. The same remains if in the end of the expression there are elements in the stack. Here is a sample implementation:

```
static void Main()
{
   string expression = "1 + (3 + 2 - (2+3)*4 - ((3+1)*(4-2)))";
   Stack<int> stack = new Stack<int>();
```

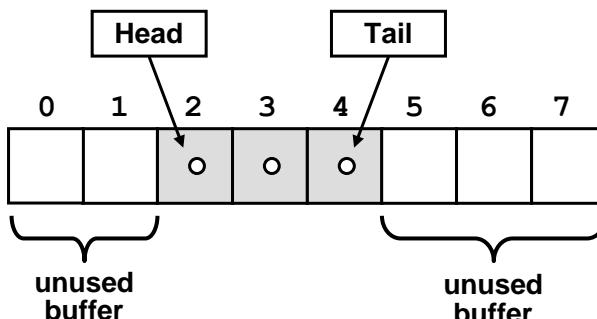
```
    bool correctBrackets = true;

    for (int index = 0; index < expression.Length; index++)
    {
        char ch = expression[index];
        if (ch == '(')
        {
            stack.Push(index);
        }
        else if (ch == ')')
        {
            if (stack.Count == 0)
            {
                correctBrackets = false;
                break;
            }
            stack.Pop();
        }
    }
    if (stack.Count != 0)
    {
        correctBrackets = false;
    }
    Console.WriteLine("Are the brackets correct? " +
        correctBrackets);
}
```

Here is how the output of the sample program looks like:

```
Are the brackets correct? True
```

Queue

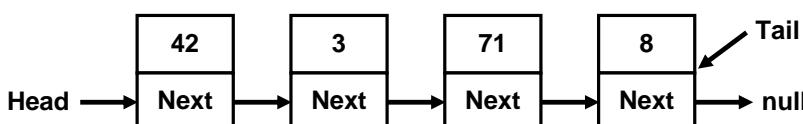
The **"queue" data structure** is created to model queues, for example a queue of waiting for printing documents, waiting processes to access a common resource, and others. Such queues are very convenient and are naturally modeled via the structure "queue". In queues we can add elements only on the back and retrieve elements only at the front.

For instance, we would like to buy a ticket for a concert. If we go earlier, we are going to buy earlier a ticket. If we are late, we will have to **go at the end of the queue and wait for everyone who has come earlier**. This behavior is analogical for the objects in ADT queue.

Abstract Data Type "Queue"

The **abstract data structure "queue"** satisfies the behavior "**first in – first out**" (**FIFO**). Elements added to the queue are appended at the end of the queue, and when elements are extracted, they are taken from the beginning of the queue (in the order they were added). Thus the queue behaves like a **list with two ends** (head and tail), just like the queues for tickets.

Like with the lists, the **ADT queue** could be implemented **statically** (as resizable array) and **dynamically** (as pointer-based linked list).

Static Queue (Array-Based Implementation)

In the **static queue** we could use an **array** for keeping the elements. When adding an element, it is inserted at the index, which follows the end of queue. After that the end points at the newly added element. When removing an element, we take the element, which is pointed by the head of the queue. After that the head starts to point at the next element. Thus **the queue moves to the end of the array**. When it reaches the end of the array, when adding a new element, it is inserted at the beginning of the array. That is why the implementation is called "**looped queue**", as we mentally stick the beginning and the end of the array and the queue orbits it:

Static queue keeps an **internal buffer** with bigger capacity than the actual number of elements in the queue. Like in the static list implementation, when the space allocated for the queue elements is finished, the **internal buffer grows** (usually doubles its size).

The **major operations** in the queue ADT are **enqueue** (append at the end of the queue) and **dequeue** (retrieve an element from the start of the queue).

Linked Queue (Dynamic Implementation)

The **dynamic implementation** of queue ADT looks like the implementation of the **linked list**. Like in the linked list, the elements consist of two parts – **a value and a pointer to the next element**:

However, here elements are **added at the end** of the queue (**tail**), and are **retrieved from its beginning** (**head**), while we have no permission to get or add elements at any another position.

The Queue<T> Class

In C# we use the static implementation of queue via the **Queue<T>** class. Here we could indicate the type of the elements we are going to work with, as the queue and the linked list are generic types.

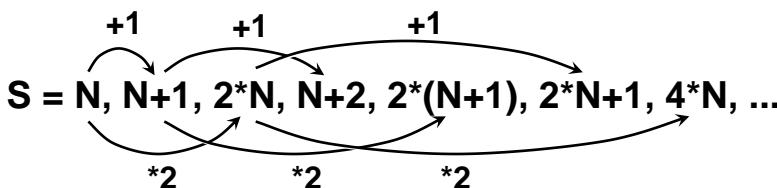
The Queue<T> – Basic Operations

Queue<T> class provides the basic operations, specific for the data structure queue. Here are some of the most frequently used:

- **Enqueue(T)** – inserts an element at the end of the queue
- **Dequeue()** – retrieves the element from the beginning of the queue and removes it
- **Peek()** – returns the element from the beginning of the queue without removing it
- **Clear()** – removes all elements from the queue
- **Contains(T)** – checks if the queue contains the element
- **Count** – returns the amount of elements in the queue

Queue Usage – Example

Let's consider a simple example. Let's create a queue and add several elements to it. After that we are going to retrieve all elements and print them on the console:

```
static void Main()
{
  Queue<string> queue = new Queue<string>();
  queue.Enqueue("Message One");
  queue.Enqueue("Message Two");
  queue.Enqueue("Message Three");
  queue.Enqueue("Message Four");

  while (queue.Count > 0)
  {
    string msg = queue.Dequeue();
    Console.WriteLine(msg);
  }
}
```

Here is how the output of the sample program looks like:

```
Message One
Message Two
Message Three
Message Four
```

You can see that the elements leave the queue in the order, in which they have entered the queue. This is because the queue is **FIFO structure (first-in, first out)**.

Sequence N, N+1, 2*N – Example

Let's consider a problem in which the usage of the data structure queue would be very useful for the implementation. Let's take the **sequence of numbers**, the elements of which are derived in the following way: the first element is N; the second element is derived by adding 1 to N; the third element – by multiplying the first element by 2 and thus we successively multiply each element by 2 and insert it at the end of the sequence, after which we add 1 to it and insert it at the end of the sequence. We could **illustrate the process** with the following figure:

As you can see, the process lies in retrieving elements from the beginning of the queue and placing others in its end. Let's see the sample implementation, in which N=3 and we search for the number of the element with value 16:

```
static void Main()
{
   int n = 3;
   int p = 16;

   Queue<int> queue = new Queue<int>();
   queue.Enqueue(n);
   int index = 0;
   Console.WriteLine("S =");
   while (queue.Count > 0)
   {
      index++;
      int current = queue.Dequeue();
      Console.WriteLine(" " + current);
      if (current == p)
      {
```

```
            Console.WriteLine();
            Console.WriteLine("Index = " + index);
            return;
        }
        queue.Enqueue(current + 1);
        queue.Enqueue(2 * current);
    }
}
```

Here is how the **output of the above program** looks like:

```
S = 3 4 6 5 8 7 12 6 10 9 16
Index = 11
```

As you can see, stack and queue are two specific data structures with strictly defined rules for the order of the elements in them. We used queue when we expected to get the elements in the order we inserted them, while we used stack when we needed the elements in reverse order.

Exercises

1. Write a program that reads from the console a sequence of positive integer numbers. The sequence ends when empty line is entered. Calculate and print the **sum** and the **average of the sequence**. Keep the sequence in **`List<int>`**.

2. Write a program, which reads from the console N integers and prints them in **reversed order**. Use the **`Stack<int>`** class.

3. Write a program that reads from the console a sequence of positive integer numbers. The sequence ends when an empty line is entered. Print the sequence **sorted in ascending order**.

4. Write a method that finds the **longest subsequence of equal numbers** in a given **`List<int>`** and returns the result as new **`List<int>`**. Write a program to test whether the method works correctly.

5. Write a program, which **removes all negative numbers** from a sequence.

   Example: array = {19, -10, 12, -6, -3, 34, -2, 5} → {19, 12, 34, 5}

6. Write a program that **removes from a given sequence all numbers that appear an odd count of times**.

   Example: array = {4, 2, 2, 5, 2, 3, 2, 3, 1, 5, 2} → {5, 3, 3, 5}

7. Write a program that finds in a given array of integers (in the range [0…1000]) **how many times each of them occurs**.

   Example: array = {3, 4, 4, 2, 3, 3, 4, 3, 2}

2 → 2 times

3 → 4 times

4 → 3 times

8. The **majorant** of an array of size N is a value that occurs in it at least N/2 + 1 times. Write a program that **finds the majorant** of given array and prints it. If it does not exist, print "The majorant does not exist!".

   Example: {2, 2, 3, 3, 2, 3, 4, 3, 3} → 3

9. We are given the following **sequence**:

   $S_1 = N$;
   $S_2 = S_1 + 1$;
   $S_3 = 2*S_1 + 1$;
   $S_4 = S_1 + 2$;
   $S_5 = S_2 + 1$;
   $S_6 = 2*S_2 + 1$;
   $S_7 = S_2 + 2$;
   …

   Using the **`Queue<T>`** class, write a program which by given **N** prints on the console the first 50 elements of the sequence.

   Example: N=2 → 2, 3, 5, 4, 4, 7, 5, 6, 11, 7, 5, 9, 6, …

10. We are given N and M and the following operations:

    N = N+1

    N = N+2

    N = N*2

    Write a program, which finds the **shortest subsequence** from the operations, which starts with **N** and ends with **M**. Use queue.

    Example: N = 5, M = 16

    Subsequence: 5 → 7 → 8 → 16

11. Implement the data structure **dynamic doubly linked list** (**`DoublyLinkedList<T>`**) – list, the elements of which have pointers both to the **next** and the **previous** elements. Implement the operations for adding, removing and searching for an element, as well as inserting an element at a given index, retrieving an element by a given index and a method, which returns an array with the elements of the list.

12. Create a **`DynamicStack<T>`** class to **implement dynamically a stack** (like a linked list, where each element knows its previous element and the stack knows its last element). Add methods for all commonly used operations like **`Push()`**, **`Pop()`**, **`Peek()`**, **`Clear()`** and **`Count`**.

13. **Implement the data structure "Deque"**. This is a specific list-like structure, similar to stack and queue, allowing to **add elements at the beginning and at the end of the structure**. Implement the operations for adding and removing elements, as well as clearing the deque. If an operation is invalid, throw an appropriate exception.

14. **Implement the structure "Circular Queue"** with array, which doubles its capacity when its capacity is full. Implement the necessary methods for adding, removing the element in succession and retrieving without removing the element in succession. If an operation is invalid, throw an appropriate exception.

15. Implement numbers **sorting** in a **dynamic linked list** without using an additional array or other data structure.

16. Using queue, implement a complete **traversal of all directories on your hard disk** and print them on the console. Implement the algorithm Breadth-First-Search (**BFS**) – you may find some articles in the internet.

17. Using queue, implement a complete **traversal of all directories on your hard disk** and print them on the console. Implement the algorithm Depth-First-Search (**DFS**) – you may find some articles in the internet.

18. We are given a **labyrinth of size N x N**. Some of the cells of the labyrinth are empty (**0**), and others are filled (**x**). We can move from an empty cell to another empty cell, if the cells are separated by a single wall. We are given a start position (*****). Calculate and fill the labyrinth as follows: in each empty cell put **the minimal distance from the start position to this cell**. If some cell cannot be reached, fill it with "**u**".

    Example:

0	0	0	x	0	x
0	x	0	x	0	x
0	*	x	0	x	0
0	x	0	0	0	0
0	0	0	x	x	0
0	0	0	x	0	x

→

3	4	5	x	u	x
2	x	6	x	u	x
1	*	x	8	x	10
2	x	6	7	8	9
3	4	5	x	x	10
4	5	6	x	u	x

Solutions and Guidelines

1. See the section "`List<T>`".

2. Use `Stack<int>`.

3. Keep the numbers in `List<T>` and finally use its `Sort()` method.

4. Use `List<int>`. Scan the list with a `for`-loop (1 … n-1) while keeping two variables: **start** and **length**. Initially start=0, length=1. At each loop iteration if the number at the left is the same as the current number,

increase length. Otherwise restart from the current cell (start=current, length=1). Remember the current start and length every time when the current length becomes better than the **current maximal length**. Finally create a new list and copy the found sequence to it.

Testing could be done through a sequence of examples and comparisons, e.g. {1} → {1}; {1, 2} → {1}; {1, 1} → {1, 1}; {1, 2, 2, 3} → {2, 2}; {1, 2, 2} → {2, 2}; {1, 1, 2} → {1, 1}; {1, 2, 2, 1, 1, 1, 2, 2, 2, 3, 3, 3} → {1, 1, 1}; {1, 2, 2, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3} → {2, 2, 2, 2}; …

5. **Use list**. Perform a left-to-right scan through all elements. If the current number is positive, add it to the result, otherwise, skip it.

6. **Slow solution:** pass through the elements with a **for**-loop. For each element **p** count how many times **p** appears in the list (with a nested **for**-loop). If it is even number of times, append **p** to the result list (which is initially empty). Finally print the result list.

   *** Fast solution:** use a **hash-table** (**Dictionary<int, int>**). With a single scan calculate **count[p]** (the number of occurrences of **p** in the input sequence) for each number **p** from the input sequence. With another single scan pass though all numbers **p** and append **p** to the result only when **count[p]** is even. Read about hash tables from the chapter "Dictionaries, Hash-Tables and Sets".

7. Make a new **array "occurrences" with size 1001**. After that scan through the list and for each number **p** increment the corresponding value of its occurrences (**occurrences[p]++**). Thus, at each index, where the value is not 0, we have an occurring number, so we print it.

8. **Use list**. **Sort the list** and you are going to get the equal numbers next to one another. Scan the array by counting the number of occurrences of each number. If up to a certain moment a number has occurred N/2+1 times, this is the **majorant** and there is no need to check further. If after position N/2+1 there is a new number (a majorant is not found until this moment), there is no need to search further – even in the case when the list is filled with the current number to the end, it will not occur N/2+1 times.

   Another solution: Use a **stack**. Scan through the elements. At each step if the element at the top of the stack is different from the next element form the input sequence, remove the element from the stack. Otherwise append the element to the stack. Finally the majorant will be in stack (if it exists). **Why?** Each time when we find any two different elements, we discard both of them. And this operation keeps the majorant the same and decreases the length of the sequence, right? If we repeat this as much times as possible, finally the stack will hold only elements with the same value – **the majorant**.

9. **Use queue**. In the beginning add N to the queue. After that take the current element M and add to the queue M+1, then 2*M + 1 and then

M+2. Repeat the same for the next element in a loop. At each step in the loop print M and if at certain point the queue size reaches 50, break the loop and finish the calculation.

10. Use the data structure **queue**. Firstly, add to the queue N. Repeat the following in a loop until M is reached: remove a number X from the queue and add 3 new elements: X * 2, X + 2 and X + 1. Do not add numbers greater than M. As optimization of the solution, **try to avoid repeating numbers** in the queue.

11. Implement **DoubleLinkedListNode<T>** class, which has fields **Previous**, **Next** and **Value**. It will hold to hold a single list node. Implement also **DoubleLinkedList<T>** class to hold the whole list.

12. Use **singly linked list** (similar to the list from the previous task, but only with a field **Previous**, without a field **Next**).

13. Just modify your implementation of **doubly-linked list** to enable adding and removing from both its head and tail. **Another solution** is to use **circular buffer** (see http://en.wikipedia.org/wiki/Circular_buffer). When the buffer is full, create a new buffer of double size and move all existing elements to it.

14. **Use array**. When you reach the last index, you need to add the next element at the beginning of the array. For the correct calculation of the indices use the **remainder from the division with the array length**. When you need to resize the array, implement it the same way like we implemented the resizing in the "Static List" section.

15. Use the simple **Bubble sort**. We start with the leftmost element by checking whether it is smaller than the next one. If it is not, we **swap their places**. Then we compare with the next element and so on and so forth, until we reach a larger element or the end of the array. We return to the start of the array and repeat the same procedure many times until we reach a moment, when we have taken sequentially all elements and no one had to be moved.

16. The **algorithm** is very easy: we start with an **empty queue**, in which we put the **root directory** (from which we start traversing). After that, until the queue is empty, we **remove the current directory** from the queue, print it on the console and **add all its subdirectories** to the queue. This way we are going to traverse the entire file system in breadth. If there are no cycles in the file system (as in Windows), the process will be finite.

17. If in the solution of the previous problem we **substitute the queue with a stack**, we are going to get **traversal in depth (DFS)**.

18. Use **Breath-First Search (BFS)** by starting from the position, marked with "*". Each unvisited adjacent to the current cell we fill with the current number + 1. We assume that the value at "*" is 0. After the queue is empty, we traverse the whole matrix and if in some of the cells we have 0, we fill it with "**u**".

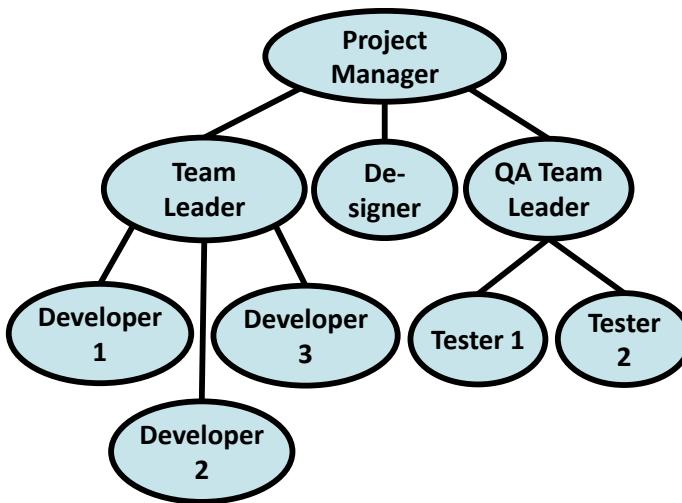
Chapter 17. Trees and Graphs

In This Chapter

In this chapter we will discuss **tree data structures**, like **trees** and **graphs**. The abilities of these data structures are really important for the modern programming. Each of this data structures is used for **building a model of real life problems**, which are efficiently solved using this model. We will explain what tree data structures are and will review their main advantages and disadvantages. We will present example implementations and problems showing their practical usage. We will focus on **binary trees**, **binary search trees** and **self-balancing binary search tree**. We will explain what **graph** is, the **types of graphs**, how to represent a graph in the memory (**graph implementation**) and where graphs are used in our life and in the computer technologies. We will see where in .NET Framework self-balancing binary search trees are implemented and how to use them.

Tree Data Structures

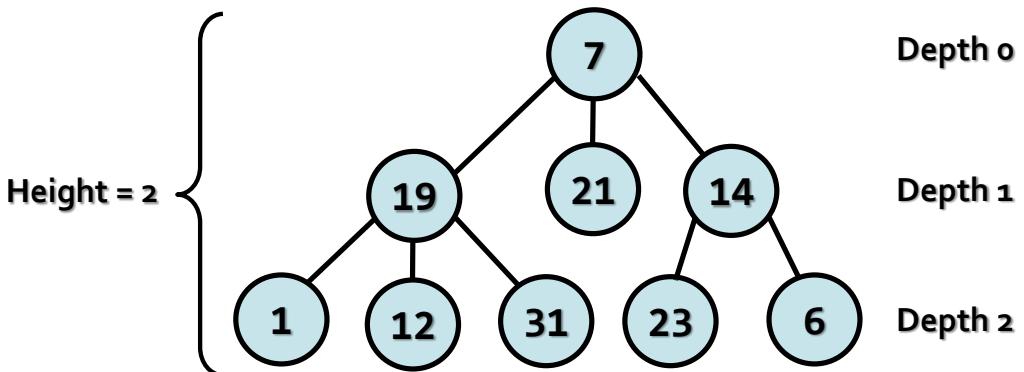
Very often we have to describe a group of real life objects, which have such relation to one another that we cannot use linear data structures for their description. In this chapter, we will give examples of such **branched structures**. We will explain their properties and the real life problems, which inspired their creation and further development.

A **tree-like data structure** or **branched data structure** consists of set of elements (nodes) which could be linked to other elements, sometimes hierarchically, sometimes not. **Trees represent hierarchies**, while **graphs represent more general relations** such as the map of city.

Trees

Trees are very often used in programming, because they naturally represent all kind of **object hierarchies** from our surroundings. Let's give an example, before we explain the trees' terminology.

Example – Hierarchy of the Participants in a Project

We have a team, responsible for the development of certain software project.

The participants in it have **manager-subordinates relations**. Our team consists of 9 teammates:

What is the information we can get from this **hierarchy**? The direct boss of the developers is the "Team Leader", but indirectly they are subordinate to the "Project Manager". The "Team Leader" is subordinate only to the "Project Manager". On the other hand "Developer 1" has no subordinates. The "Project Manager" is the highest in the hierarchy and has no manager.

The same way we can describe every participant in the project. We see that such a little figure gives us so much information.

Trees Terminology

For a better understanding of this part of the chapter we recommend to the reader at every step to draw an analogy between the abstract meaning and its practical usage in everyday life.

We will simplify the figure describing our **hierarchy**. We assume that it consists of circles and lines connecting them. For convenience we name the circles with unique numbers, so that we can easily specify about which one we are talking about.

We will call every circle a **node** and each line an **edge**. Nodes "19", "21", "14" are below node "7" and are directly connected to it. This nodes we are called

direct descendants (child nodes) of node "7", and node "7" their **parent**. The same way "1", "12" and "31" are children of "19" and "19" is their parent. Intuitively we can say that "21" is **sibling** of "19", because they are both children of "7" (the reverse is also true – "19" is sibling of "21").For "1", "12", "31", "23" and "6" node "7" precedes them in the hierarchy, so he is their indirect parent – **ancestor**, ant they are called his **descendants**.

Root is called the **node without parent**. In our example this is node "7"

Leaf is a **node without child nodes**. In our example – "1", "12", "31", "21", "23" and "6".

Internal nodes are the nodes, which are **not leaf or root** (all nodes, which have parent and at least one child). Such nodes are "19" and "14".

Path is called a **sequence of nodes connected with edges**, in which there is no repetition of nodes. Example of path is the sequence "1", "19", "7" and "21". The sequence "1", "19" and "23" is not a path, because "19" and "23" are not connected.

Path length is the number of edges, connecting the sequence of nodes in the path. Actually it is equal to the **number of nodes in the path minus 1**. The length of our example for path ("1", "19", "7" and "21") is three.

Depth of a node we will call the **length of the path from the root to certain node**. In our example "7" as root has depth zero, "19" has depth one and "23" – depth two.

Here is the definition about tree:

Tree – a **recursive data structure**, which consists of **nodes, connected with edges**. The following statements are true for trees:

- Each node can have **0 or more direct descendants** (children).
- Each node has **at most one parent**. There is only one special node without parent – **the root** (if the tree is not empty).
- All nodes are **reachable from the root** – there is a path from the root to each node in the tree.

We can give more simple definition of tree: **a node is a tree and this node can have zero or more children, which are also trees**.

Height of tree – is the **maximum depth** of all its nodes. In our example the tree height is 2.

Degree of node we call the **number of direct children** of the given node. The degree of "19" and "7" is three, but the degree of "14" is two. The leaves have degree zero.

Branching factor is the **maximum of the degrees of all nodes** in the tree. In our example the maximum degree of the nodes is 3, so the branching factor is 3.

Tree Implementation – Example

Now we will see **how to represent trees as data structure** in programming. We will implement a tree dynamically. Our tree will contain numbers inside its **nodes**, and each node will have a **list of zero or more children**, which are trees too (following our recursive definition).

Each node is recursively defined using itself. Each node of the tree (**TreeNode<T>**) contains a list of children, which are nodes (**TreeNode<T>**). The tree itself is another class **Tree<T>** which can be empty or can have a root node. **Tree<T>** implements basic operations over trees like construction and traversal.

Let's have a look at the source code of our **dynamic tree representation**:

```
using System;
using System.Collections.Generic;

/// <summary>Represents a tree node</summary>
/// <typeparam name="T">the type of the values in nodes
/// </typeparam>
public class TreeNode<T>
{
   // Contains the value of the node
   private T value;

   // Shows whether the current node has a parent or not
   private bool hasParent;

   // Contains the children of the node (zero or more)
   private List<TreeNode<T>> children;

   /// <summary>Constructs a tree node</summary>
   /// <param name="value">the value of the node</param>
   public TreeNode(T value)
   {
      if (value == null)
      {
         throw new ArgumentNullException(
            "Cannot insert null value!");
      }
      this.value = value;
      this.children = new List<TreeNode<T>>();
   }

   /// <summary>The value of the node</summary>
   public T Value
```

```
{
   get
   {
      return this.value;
   }
   set
   {
      this.value = value;
   }
}

/// <summary>The number of node's children</summary>
public int ChildrenCount
{
   get
   {
      return this.children.Count;
   }
}

/// <summary>Adds child to the node</summary>
/// <param name="child">the child to be added</param>
public void AddChild(TreeNode<T> child)
{
   if (child == null)
   {
      throw new ArgumentNullException(
         "Cannot insert null value!");
   }

   if (child.hasParent)
   {
      throw new ArgumentException(
         "The node already has a parent!");
   }

   child.hasParent = true;
   this.children.Add(child);
}

/// <summary>
/// Gets the child of the node at given index
/// </summary>
/// <param name="index">the index of the desired child</param>
```

```
    /// <returns>the child on the given position</returns>
    public TreeNode<T> GetChild(int index)
    {
        return this.children[index];
    }
}

/// <summary>Represents a tree data structure</summary>
/// <typeparam name="T">the type of the values in the
/// tree</typeparam>
public class Tree<T>
{
    // The root of the tree
    private TreeNode<T> root;

    /// <summary>Constructs the tree</summary>
    /// <param name="value">the value of the node</param>
    public Tree(T value)
    {
        if (value == null)
        {
            throw new ArgumentNullException(
                "Cannot insert null value!");
        }

        this.root = new TreeNode<T>(value);
    }

    /// <summary>Constructs the tree</summary>
    /// <param name="value">the value of the root node</param>
    /// <param name="children">the children of the root
    /// node</param>
    public Tree(T value, params Tree<T>[] children)
        : this(value)
    {
        foreach (Tree<T> child in children)
        {
            this.root.AddChild(child.root);
        }
    }

    /// <summary>
    /// The root node or null if the tree is empty
    /// </summary>
```

```
    public TreeNode<T> Root
    {
        get
        {
            return this.root;
        }
    }

    /// <summary>Traverses and prints tree in
    /// Depth-First Search (DFS) manner</summary>
    /// <param name="root">the root of the tree to be
    /// traversed</param>
    /// <param name="spaces">the spaces used for
    /// representation of the parent-child relation</param>
    private void PrintDFS(TreeNode<T> root, string spaces)
    {
        if (this.root == null)
        {
            return;
        }

        Console.WriteLine(spaces + root.Value);

        TreeNode<T> child = null;
        for (int i = 0; i < root.ChildrenCount; i++)
        {
            child = root.GetChild(i);
            PrintDFS(child, spaces + "   ");
        }
    }

    /// <summary>Traverses and prints the tree in
    /// Depth-First Search (DFS) manner</summary>
    public void TraverseDFS()
    {
        this.PrintDFS(this.root, string.Empty);
    }
}

/// <summary>
/// Shows a sample usage of the Tree<T> class
/// </summary>
public static class TreeExample
{
```

```
  static void Main()
  {
    // Create the tree from the sample
    Tree<int> tree =
      new Tree<int>(7,
        new Tree<int>(19,
          new Tree<int>(1),
          new Tree<int>(12),
          new Tree<int>(31)),
        new Tree<int>(21),
        new Tree<int>(14,
          new Tree<int>(23),
          new Tree<int>(6))
      );

    // Traverse and print the tree using Depth-First-Search
    tree.TraverseDFS();

    // Console output:
    // 7
    //   19
    //      1
    //      12
    //      31
    //   21
    //   14
    //      23
    //      6
  }
}
```

How Does Our Implementation Work?

Let's discuss the given code a little. In our example we have a class **Tree<T>**, which implements **the actual tree**. We also have a class **TreeNode<T>**, which represents a **single node of the tree**.

The functions associated with node, like creating a node, adding a child node to this node, and getting the number of children, are implemented at the level of **TreeNode<T>**.

The rest of the functionality (traversing the tree for example) is implemented at the level of **Tree<T>**. Logically dividing the functionality between the two classes makes our implementation more flexible.

The reason we divide the implementation in two classes is that some operations are **typical for each separate node** (adding a child for example),

while others are **about the whole tree** (searching a node by its number). In this variant of the implementation, the tree is a class that knows its root and each node knows its children. In this implementation we can have an empty tree (when **root = null**).

Here are some details about the **TreeNode<T>** implementation. Each node of the tree consists of private field **value** and a list of children – **children**. The list of children consists of elements of the same type. That way each node contains a **list of references to its direct children**. There are also public properties for accessing the values of the fields of the node. The methods that can be called from code outside the class are:

- **AddChild(TreeNode<T> child)** – adds a child
- **TreeNode<T> GetChild(int index)** – returns a child by given index
- **ChildrenCount** – returns the number of children of certain node

To satisfy the condition that every node has only one parent we have defined private field **hasParent**, which determines whether this node has parent or not. This information is used only inside the class and we need it in the **AddChild(Tree<T> child)** method. Inside this method we check whether the node to be added already has parent and if so we throw and exception, saying that this is impossible.

In the class **Tree<T>** we have only one **get** property **TreeNode<T> Root**, which returns the root of the tree.

Depth-First-Search (DFS) Traversal

In the class **Tree<T>** is implemented the method **TraverseDFS()**, that calls the private method **PrintDFS(TreeNode<T> root, string spaces)**, which traverses the tree in depth and prints on the standard output its elements in tree layout using right displacement (adding spaces).

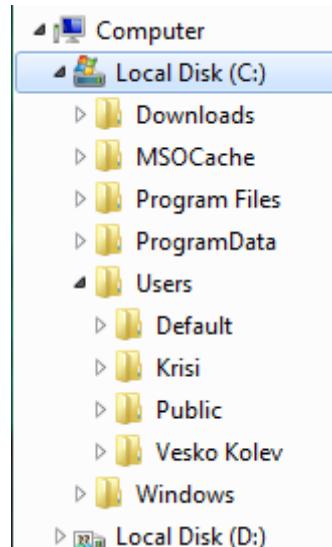
The **Depth-First-Search algorithm** aims to visit each of the tree nodes exactly one. Such a visit of all nodes is called **tree traversal**. There are multiple algorithms to traverse a tree but in this chapter we will discuss only two of them: **DFS** (depth-first search) and **BFS** (breadth-first search).

The **DFS algorithm** starts from a given node and goes as deep in the tree hierarchy as it can. When it reaches a node, which has no children to visit or all have been visited, it returns to the previous node. We can describe the depth-first search algorithm by the following simple steps:

1. **Traverse the current node** (e.g. print it on the console or process it in some way).
2. Sequentially **traverse recursively each of the current nodes' child nodes** (traverse the sub-trees of the current node). This can be done by a recursive call to the same method for each child node.

Creating a Tree

We to make **creating a tree** easier we defined a **special constructor**, which takes for input parameters **a node value and a list of its sub-trees**. That allows us to give any number of arguments of type **`Tree<T>`** (sub-trees). We used exactly the same constructor for creating the example tree.

Traverse the Hard Drive Directories

Let's start with another example of tree: **the file system**. Have you noticed that the directories on your hard drive are actually a **hierarchical structure**, which is a tree? We have **folders** (tree nodes) which may have **child folders and files** (which both are also tree nodes).

You can think of many real life examples, where trees are used, right?

Let's get a more detailed view of Windows file system. As we know from our everyday experience, we create **folders** on the hard drive, which can contain **subfolders** and **files**. Subfolders can also contain subfolders and so on until you reach certain max depth limit.

The directory tree of the file system is accessible through the build in .NET functionality: the class **`System.IO.DirectoryInfo`**. It is not present as a data structure, but we can get the subfolders and files of every directory, so we can **traverse the file system tree** by using a standard tree traversal algorithm, such as Depth-First Search (DFS).

Below we can see what the typical directory tree in Windows looks like:

Recursive DFS Traversal of the Directories

The next example illustrates how we can recursively **traverse recursively the tree structure of given folder** (using Depth-First-Search) and print on the standard output its content:

DirectoryTraverserDFS.cs

```
using System;
using System.IO;

/// <summary>
/// Sample class, which traverses recursively given directory
/// based on the Depth-First-Search (DFS) algorithm
/// </summary>
public static class DirectoryTraverserDFS
{
    /// <summary>
    /// Traverses and prints given directory recursively
    /// </summary>
    /// <param name="dir">the directory to be traversed</param>
    /// <param name="spaces">the spaces used for representation
    /// of the parent-child relation</param>
    private static void TraverseDir(DirectoryInfo dir,
        string spaces)
    {
        // Visit the current directory
        Console.WriteLine(spaces + dir.FullName);

        DirectoryInfo[] children = dir.GetDirectories();

        // For each child go and visit its sub-tree
        foreach (DirectoryInfo child in children)
        {
            TraverseDir(child, spaces + "  ");
        }
    }

    /// <summary>
    /// Traverses and prints given directory recursively
    /// </summary>
    /// <param name="directoryPath">the path to the directory
    /// which should be traversed</param>
    static void TraverseDir(string directoryPath)
    {
        TraverseDir(new DirectoryInfo(directoryPath),
            string.Empty);
    }

    static void Main()
```

```
    {
        TraverseDir("C:\\");
    }
}
```

As we can see the recursive traversal algorithm of the content of the directory is the same as the one we used for our tree.

Here we can see part of the result of the traversal:

```
C:\
  C:\Config.Msi
  C:\Documents and Settings
    C:\Documents and Settings\Administrator
      C:\Documents and Settings\Administrator\.ARIS70
      C:\Documents and Settings\Administrator\.jindent
      C:\Documents and Settings\Administrator\.nbi
        C:\Documents and Settings\Administrator\.nbi\downloads
        C:\Documents and Settings\Administrator\.nbi\log
        C:\Documents and Settings\Administrator\.nbi\cache
        C:\Documents and Settings\Administrator\.nbi\tmp
        C:\Documents and Settings\Administrator\.nbi\wd
      C:\Documents and Settings\Administrator\.netbeans
        C:\Documents and Settings\Administrator\.netbeans\6.0
…
```

Note that the above program may crash with **UnauthorizedAccessException** in case you do not have access permissions for some folders on the hard disk. This is typical for some Windows installations so you could start the traversal from another directory to play with it, e.g. from "**C:\Windows\assembly**".

Breath-First-Search (BFS)

Let's have a look at another way of traversing trees. **Breath-First-Search (BFS)** is an algorithm for traversing branched data structures (like trees and graphs). The BFS algorithm first traverses the start node, then all its direct children, then their direct children and so on. This approach is also known as the **wavefront traversal**, because it looks like the waves caused by a stone thrown into a lake.

The **Breath-First-Search (BFS) algorithm** consists of the following steps:

1. Enqueue the start node in queue **Q**.
2. While **Q** is not empty repeat the following two steps:
   - Dequeue the next node **v** from **Q** and print it.
   - Add all children of **v** in the queue.

The BFS algorithm is very simple and always traverses first the nodes that are closest to the start node, and then the more distant and so on until it reaches the furthest. The BFS algorithm is very widely used in problem solving, e.g. for **finding the shortest path in a labyrinth**.

A sample **implementation of BFS algorithms** that prints all folders in the file system is given below:

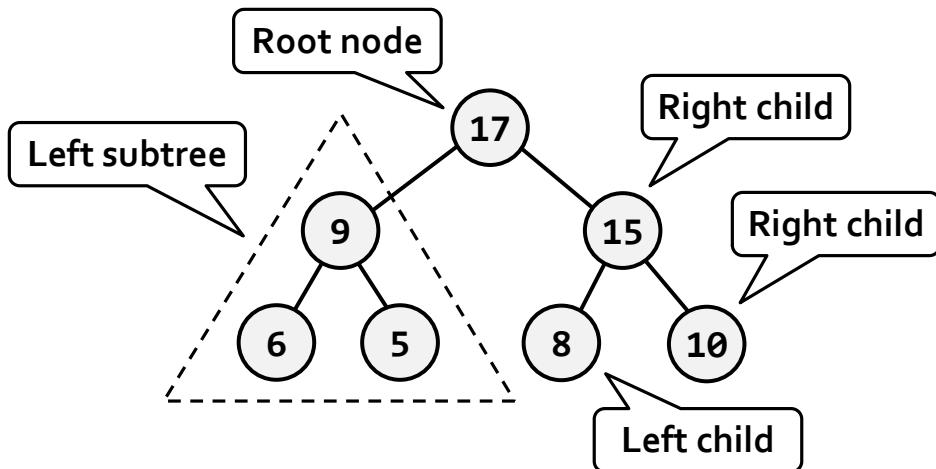
DirectoryTraverserBFS.cs

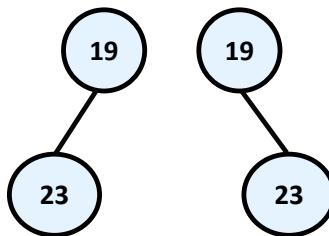
```
using System;
using System.Collections.Generic;
using System.IO;

/// <summary>
/// Sample class, which traverses given directory
/// based on the Breath-First-Search (BFS) algorithm
/// </summary>
public static class DirectoryTraverserBFS
{
   /// <summary>
   /// Traverses and prints given directory with BFS
   /// </summary>
   /// <param name="directoryPath">the path to the directory
   /// which should be traversed</param>
   static void TraverseDir(string directoryPath)
   {
      Queue<DirectoryInfo> visitedDirsQueue =
         new Queue<DirectoryInfo>();
      visitedDirsQueue.Enqueue(new DirectoryInfo(directoryPath));
      while (visitedDirsQueue.Count > 0)
      {
         DirectoryInfo currentDir = visitedDirsQueue.Dequeue();
         Console.WriteLine(currentDir.FullName);

         DirectoryInfo[] children = currentDir.GetDirectories();
         foreach (DirectoryInfo child in children)
         {
            visitedDirsQueue.Enqueue(child);
         }
      }
   }

   static void Main()
   {
      TraverseDir(@"C:\");
```

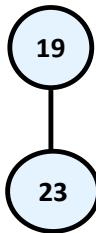
```
    }
}
```

If we start the program to traverse our local hard disk, we will see that the BFS first visits the directories closest to the root (depth 1), then the folders at depth 2, then depth 3 and so on. Here is a sample output of the program:

```
C:\
C:\Config.Msi
C:\Documents and Settings
C:\Inetpub
C:\Program Files
C:\RECYCLER
C:\System Volume Information
C:\WINDOWS
C:\wmpub
C:\Documents and Settings\Administrator
C:\Documents and Settings\All Users
C:\Documents and Settings\Default User
…
```

Binary Trees

In the previous section we discussed the basic structure of a tree. In this section we will have a look at a specific type of tree – **binary tree**. This type of tree turns out to be very useful in programming. The terminology for trees is also valid about binary trees. Despite that below we will give some specific explanations about thus structure.

Binary Tree – a tree, which nodes have a **degree equal or less than 2** or we can say that it is a tree with **branching degree of 2**. Because every node's children are at most 2, we call them **left child** and **right child**. They are the roots of the **left sub-tree** and the **right sub-tree** of their parent node. Some nodes may have only left or only right child, not both. Some nodes may have no children and are called **leaves**.

Binary tree can be **recursively** defined as follows: **a single node is a binary tree and can have left and right children which are also binary trees**.

Binary Tree – Example

Here we have an example of **binary tree**. The nodes are again named with some numbers. An the figure we can see the **root of the tree** – "14", the **left sub-tree** (with root 19) and the **right sub-tree** (with root 15) and a **right** and **left child** – "3" and "21".

We have to note that there is one very big difference in the definition of binary tree from the definition of the classical tree – the **order of the children** of each node. The next example will illustrate that difference:

On this figure above two totally different **binary trees** are illustrated – the first one has root "19" and its **left child** "23" and the second root "19" and **right child** "23". If that was an ordinary tree they would have been the same. That's why such **tree** we would illustrate the following way:

	Remember! Although we take binary trees as a special case of a tree structure, we have to notice that the condition for particular order of children nodes makes them a completely different structure.

Binary Tree Traversal

The **traversal of binary tree** is a classic problem which has classical solutions. Generally there are few ways to traverse a binary tree recursively:

- **In-order (Left-Root-Right)** – the traversal algorithm first traverses the left sub-tree, then the root and last the left sub-tree. In our example the sequence of such traversal is: "23", "19", "10", "6", "21", "14", "3", "15".
- **Pre-order (Root-Left-Right)** – in this case the algorithm first traverses the root, then the left sub-tree and last the right sub-tree. The result of such traversal in our example is: "14", "19", "23", "6", "10", "21", "15", "3".
- **Post-order (Left-Right-Root)** – here we first traverse the left sub-tree, then the right one and last the root. The result after the traversal is: "23", "10", "21", "6", "19", "3", "15", "14".

Recursive Traversal of Binary Tree – Example

The next example shows an implementation of binary tree, which we will traverse using the **in-order recursive scheme**.

```
using System;
using System.Collections.Generic;

/// <summary>Represents a binary tree</summary>
/// <typeparam name="T">Type of values in the tree</typeparam>
public class BinaryTree<T>
{
  /// <summary>The value stored in the curent node</summary>
  public T Value { get; set; }

  /// <summary>The left child of the current node</summary>
  public BinaryTree<T> LeftChild { get; private set; }

  /// <summary>The right child of the current node</summary>
  public BinaryTree<T> RightChild { get; private set; }

  /// <summary>Constructs a binary tree</summary>
  /// <param name="value">the value of the tree node</param>
  /// <param name="leftChild">the left child of the tree</param>
  /// <param name="rightChild">the right child of the tree
  /// </param>
  public BinaryTree(T value,
    BinaryTree<T> leftChild, BinaryTree<T> rightChild)
  {
    this.Value = value;
    this.LeftChild = leftChild;
    this.RightChild = rightChild;
  }
```

```
    /// <summary>Constructs a binary tree with no children
    /// </summary>
    /// <param name="value">the value of the tree node</param>
    public BinaryTree(T value) : this(value, null, null)
    {
    }

    /// <summary>Traverses the binary tree in pre-order</summary>
    public void PrintInOrder()
    {
        // 1. Visit the left child
        if (this.LeftChild != null)
        {
            this.LeftChild.PrintInOrder();
        }

        // 2. Visit the root of this sub-tree
        Console.Write(this.Value + " ");

        // 3. Visit the right child
        if (this.RightChild != null)
        {
            this.RightChild.PrintInOrder();
        }
    }
}

/// <summary>
/// Demonstrates how the BinaryTree<T> class can be used
/// </summary>
public class BinaryTreeExample
{
    static void Main()
    {
        // Create the binary tree from the sample
        BinaryTree<int> binaryTree =
            new BinaryTree<int>(14,
                new BinaryTree<int>(19,
                    new BinaryTree<int>(23),
                    new BinaryTree<int>(6,
                        new BinaryTree<int>(10),
                        new BinaryTree<int>(21))),
                new BinaryTree<int>(15,
```

```
                new BinaryTree<int>(3),
                null));

        // Traverse and print the tree in in-order manner
        binaryTree.PrintInOrder();
        Console.WriteLine();

        // Console output:
        // 23 19 10 6 21 14 3 15
    }
}
```

How Does the Example Work?

This implementation of binary tree is slightly different from the one of the ordinary tree and is significantly simplified.

We have a **recursive class definition `BinaryTree<T>`**, which holds a **value** and **left** and **right child nodes** which are of the same type **`BinaryTree<T>`**. We have **exactly two child nodes** (left and right) instead of list of children.

The method **`PrintInOrder()`** works recursively using the DFS algorithm. It traverses each node in "in-order" (first the left child, then the node itself, then the right child). The DFS traversal algorithm performs the following steps:

1. Recursive call to **traverse the left sub-tree** of the given node.
2. **Traverse the node itself** (print its value).
3. Recursive call to **traverse the right sub-tree**.

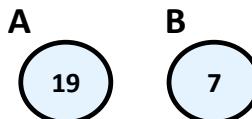
We highly recommend the reader to try and modify the algorithm and the source code of the given example to implement the other types of binary tree traversal of binary (**pre-order** and **post-order**) and see the difference.

Ordered Binary Search Trees

Till this moment we have seen how we can build **traditional and binary trees**. These structures are very summarized in themselves and it will be difficult for us to use them for a bigger project. Practically, in computer science special and programming variants of binary and ordinary trees are used that have certain special characteristics, like order, minimal depth and others. Let's review **the most important trees used in programming**.

As examples for a useful properties we can give the ability to quickly search of an element by given value (**Red-Black tree**); order of the elements in the tree (**ordered search trees**); balanced depth (**balanced trees**); possibility to store an ordered tree in a persistent storage so that searching of an element to be fast with as little as possible read operations (**B-tree**), etc.

In this chapter we will take a look at a more specific class of binary trees – **ordered trees**. They use one often met property of the nodes in the binary trees – **unique identification key** in every node. Important property of these keys is that they are comparable. Important kind of ordered trees are the so called "**balanced search trees**".

Comparability between Objects

Before continuing, we will introduce the following definition, which we will need for the further exposure.

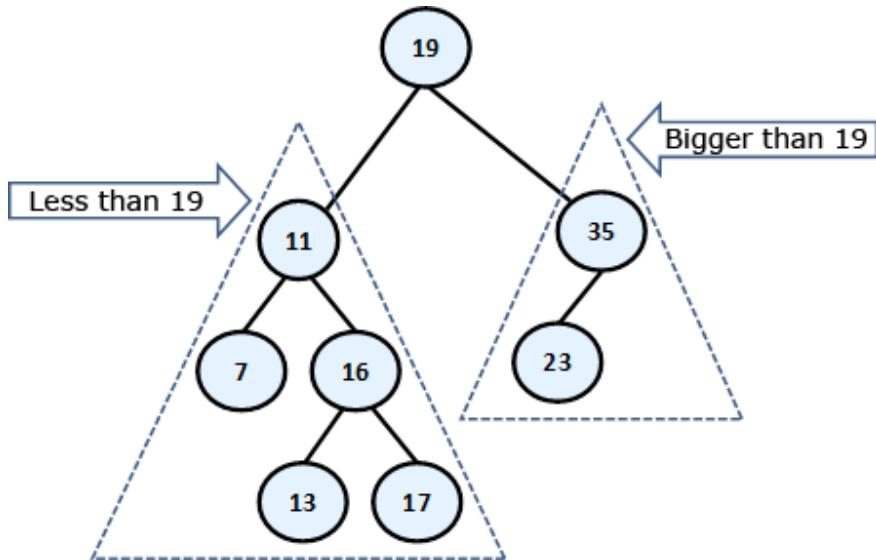
Comparability – we call two objects A and B **comparable**, if **exactly one** of following three dependencies exists:

- "A is less than B"
- "A is bigger than B"
- "A is equal to B"

Similarly we will call two **keys A and B comparable**, if exactly one of the following three possibilities is true: A < B, A > B or A = B.

The nodes of a tree can contain different fields but we can think about only their unique keys, which we want to be comparable. Let's give an example. We have two specific nodes A and B:

In this case, the keys of A and B hold the integer numbers 19 and 7. From Mathematics we know that the integer numbers (unlike the complex numbers) are **comparable**, which according the above reasoning give us the right to use them as keys. That's why we can say that "A is bigger than B", because "19 is bigger than 17".

Please notice! In this case the numbers depicted on the nodes are their unique identification keys and not like before, just some numbers.

And we arrive to the definition of the **ordered binary search tree**:

Ordered Binary Tree (binary search tree) is a binary tree, in which every node has a unique key, every two of the keys are comparable and the tree is organized in a way that for every node the following is satisfied:

- All keys in **the left sub-tree** are **smaller** than its key.
- All keys in **the right sub-tree** are **bigger** than its key.

Properties of the Ordered Binary Search Trees

On the figure below we have given an **example of an ordered binary search tree**. We will use this example, to give some important properties of the binary tree's order:

By definition we know that **the left sub-tree** of every node consists only of **elements, which are smaller than itself**, while in the **right sub-tree** there are only **bigger elements**. This means that if we want to find a given element, starting from the root, either we have found it or should search it respectively in its left or its right sub-tree, which will save unnecessary comparisons. For example, if we search 23 in our tree, we are not going to search for it in the left sub-tree of 19, because 23 is not there for sure (23 is bigger than 19, so eventually it is in the right sub-tree). This saves us 5 unnecessary comparisons with each of the left sub-tree elements, but if we were using a linked list, we would have to make these 5 comparisons.

From the elements' order follows that **the smallest** element in the tree is **the leftmost successor of the root**, if there is such **or the root** itself, if it does not have a left successor. In our example this is the minimal element 7 and the maximal – 35. Next useful property from this is, that every single element from the left sub-tree of given node is smaller than every single element from the right sub-tree of the same node.

Ordered Binary Search Trees – Example

The next example shows a simple implementation of a **binary search tree**. Our point is to suggest methods for adding, searching and removing an element in the tree. For every single operation from the above, we will give an explanation in details. Note that our binary search tree **is not balanced** and may have poor performance in certain circumstances.

Ordered Binary Search Trees: Implementation of the Nodes

Just like before, now we will define an internal class, which will describe a **node's structure**. Thus we will clearly distinguish and encapsulate the structure of a **node**, which our **tree** will contain within itself. This separate class **BinaryTreeNode<T>** that we have defined as internal is visible only in the ordered tree's class.

Here is its definition:

BinaryTreeNode.cs

```
…
/// <summary>Represents a binary tree node</summary>
/// <typeparam name="T">Specifies the type for the values
/// in the nodes</typeparam>
internal class BinaryTreeNode<T> :
  IComparable<BinaryTreeNode<T>> where T : IComparable<T>
{
  // Contains the value of the node
  internal T value;

  // Contains the parent of the node
  internal BinaryTreeNode<T> parent;

  // Contains the left child of the node
  internal BinaryTreeNode<T> leftChild;

  // Contains the right child of the node
  internal BinaryTreeNode<T> rightChild;

  /// <summary>Constructs the tree node</summary>
  /// <param name="value">The value of the tree node</param>
  public BinaryTreeNode(T value)
  {
    if (value == null)
    {
      // Null values cannot be compared -> do not allow them
      throw new ArgumentNullException(
        "Cannot insert null value!");
    }

    this.value = value;
    this.parent = null;
    this.leftChild = null;
    this.rightChild = null;
```

```
    }

    public override string ToString()
    {
        return this.value.ToString();
    }

    public override int GetHashCode()
    {
        return this.value.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        BinaryTreeNode<T> other = (BinaryTreeNode<T>)obj;
        return this.CompareTo(other) == 0;
    }

    public int CompareTo(BinaryTreeNode<T> other)
    {
        return this.value.CompareTo(other.value);
    }
}
…
```

Let's have a look to the proposed code. Still in the name of the structure, which we are considering – "**ordered search tree**", we are talking about order and we can achieve this order **only** if we have **comparability** among the elements in the tree.

Comparability between Objects in C#

What does "**comparability between objects**" mean for us as developers? It means that we must somehow oblige everyone who uses our data structure, to create it passing it a **type, which is comparable**.

In C# the sentence "**type, which is comparable**" will sound like this:

```
T : IComparable<T>
```

The **interface `IComparable<T>`**, located in the namespace **`System`**, specifies the method **`CompareTo(T obj)`**, which returns a negative integer number, zero or a positive integer number respectively if the current object is less, equal or bigger than the one which is given to the method for comparing. Its definition looks approximately like this:

```
public interface IComparable<T>
{
  /// <summary>Compares the current object with another
  /// object of the same type.</summary>
  int CompareTo(T other);
}
```

On one hand, the implementation of this interface by given class ensures us that its instances are comparable (more about interfaces in OOP can be found in the "Interfaces" section of the "Defining Classes" chapter).

On the other hand, we need those nodes, described by **BinaryTreeNode<T>** class to be comparable between them too. That is why it implements **IComparable<T>** too. As it is shown in the code, the implementation of **IComparable<T>** to the **BinaryTreeNode<T>** class calls the type **T**'s implementation internally.

In the code we have also implemented the methods **Equals(Object obj)** and **GetHashCode()** too. A good (recommended) practice is these two methods to be consistent in their behavior, i.e. when two objects are the same, then their hash-code is the same. As we will see in the chapter about hash tables, the opposite is not necessary at all. Similarly – the expected behavior of the **Equals(Object obj)** is to return **true**, exactly when **CompareTo(T obj)** returns **0**.

It's recommended to sync the work of Equals(Object obj), CompareTo(T obj) and GetHashCode() methods. This is their expected behavior and it will save you a lot of hard to find problems.

Till now, we have discussed the methods, suggested by our class. Now let's see what fields it provides. They are respectively for **value** (the key) of type **T** parent – **parent**, left and right successor – **leftChild** and **rightChild**. The last three are of the type of the defining them class – **BinaryTreeNode**

Ordered Binary Trees – Implementation of the Main Class

Now, we go to the implementation of the class, describing an ordered binary tree – **BinarySearchTree<T>**. The tree by itself as a structure consists of a root node of type **BinaryTreeNode<T>**, which contains internally its successors – **left** and **right**. Internally they also contain their successors, thus recursively down until it reaches the leaves.

An important thing is the definition **BinarySearchTree<T> where T : IComparable<T>**. This constraint of the type **T** is necessary because of the requirement of our internal class, which works only with types, implementing **IComparable<T>**. Due to this restriction we can use **BinarySearchTree<int>** and **BinarySearchTree<string>**, but cannot use **BinarySearchTree<int[]>**

and **BinarySearchTree<StreamReader>**, because **int[]** and **StreamReader** are not comparable, while **int** and **string** are.

BinarySearchTree.cs

```
public class BinarySearchTree<T> where T : IComparable<T>
{
  /// <summary>
  /// Represents a binary tree node
  /// </summary>
  /// <typeparam name="T">The type of the nodes</typeparam>
  internal class BinaryTreeNode<T> :
    IComparable<BinaryTreeNode<T>> where T : IComparable<T>
  {
    // …
    // … The implementation from above comes here!!! …
    // …
  }

  /// <summary>
  /// The root of the tree
  /// </summary>
  private BinaryTreeNode<T> root;

  /// <summary>
  /// Constructs the tree
  /// </summary>
  public BinarySearchTree()
  {
    this.root = null;
  }

  // …
  // … The implementation of tree operations come here!!! …
  // …
}
```

As we mentioned above, now we will examine the following operations:

- **insert** an element;
- **searching** for an element;
- **removing** an element.

Inserting an Element

Inserting (or adding) an element in a binary search tree means to put a new element somewhere in the tree so that the tree must **stay ordered**. Here is the algorithm: if the tree is empty, we add the new element as a root. Otherwise:

- If the element is **smaller than the root**, we call recursively the same method to add the element in the left sub-tree.
- If the element is **bigger than the root**, we call recursively to the same method to add the element in the right sub-tree.
- If the element is **equal to the root**, we don't do anything and exit from the recursion.

We can clearly see how the **algorithm for inserting a node**, conforms to the rule "elements in the left sub-tree are less than the root and the elements in the right sub-tree are bigger than the root". Here is a sample implementation of this method. You should notice that in the addition there is a reference to the parent, which is supported because the parent must be changed too.

```
/// <summary>Inserts new value in the binary search tree
/// </summary>
/// <param name="value">the value to be inserted</param>
public void Insert(T value)
{
  this.root = Insert(value, null, root);
}

/// <summary>
/// Inserts node in the binary search tree by given value
/// </summary>
/// <param name="value">the new value</param>
/// <param name="parentNode">the parent of the new node</param>
/// <param name="node">current node</param>
/// <returns>the inserted node</returns>
private BinaryTreeNode<T> Insert(T value,
    BinaryTreeNode<T> parentNode, BinaryTreeNode<T> node)
{
  if (node == null)
  {
    node = new BinaryTreeNode<T>(value);
    node.parent = parentNode;
  }
  else
  {
    int compareTo = value.CompareTo(node.value);
```

```
        if (compareTo < 0)
        {
            node.leftChild =
                Insert(value, node, node.leftChild);
        }
        else if (compareTo > 0)
        {
            node.rightChild =
                Insert(value, node, node.rightChild);
        }
    }

    return node;
}
```

Searching for an Element

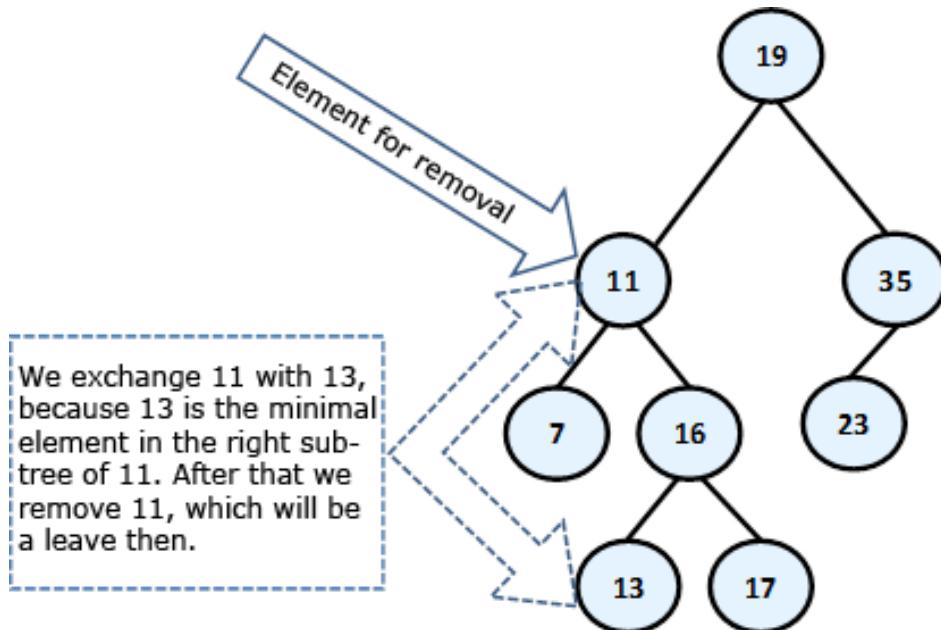
Searching in a binary search tree is an operation which is more intuitive. In the sample code we have shown how the search of an element can be done without recursion and with iteration instead. The algorithm starts with element **node**, pointing to the root. After that we do the following:

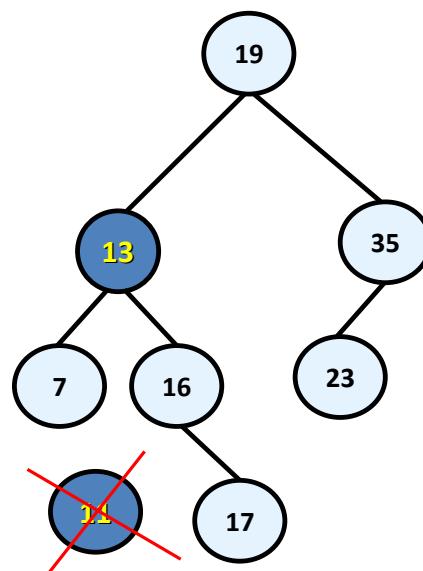
- If the element is **equal to node**, we have found the searched element and return it.
- If the element is **smaller than node**, we assign to **node** its left successor, i.e. we continue the searching in the left sub-tree.
- If the element is **bigger than node**, we assign to **node** its right successor, i.e. we continue the searching in the right sub-tree.

At the end, the algorithm **returns the found node or null** if there is no such node in the tree. Additionally we define a Boolean method that checks if certain value belongs to the tree. Here is the sample code:

```
/// <summary>Finds a given value in the tree and
/// return the node which contains it if such exsists
/// </summary>
/// <param name="value">the value to be found</param>
/// <returns>the found node or null if not found</returns>
private BinaryTreeNode<T> Find(T value)
{
    BinaryTreeNode<T> node = this.root;
    while (node != null)
    {
        int compareTo = value.CompareTo(node.value);
        if (compareTo < 0)
```

```
        {
            node = node.leftChild;
        }
        else if (compareTo > 0)
        {
            node = node.rightChild;
        }
        else
        {
            break;
        }
    }

    return node;
}

/// <summary>Returns whether given value exists in the tree
/// </summary>
/// <param name="value">the value to be checked</param>
/// <returns>true if the value is found in the tree</returns>
public bool Contains(T value)
{
    bool found = this.Find(value) != null;
    return found;
}
```

Removing an Element

Removing is the most complicated operation from the basic binary search tree operations. After it the tree must **keep its order**.

The first step before we remove an element from the tree is to **find it**. We already know how it happens. After that, we have 3 cases:

- If the node is a **leaf** – we point its parent's reference to **null**. If the element has no parent, it means that it is a root and we just remove it.
- If the **node has only one sub-tree** – left or right, it is replacing with the root of this sub-tree.
- The **node has two sub-trees**. Then we have to find the smallest node in the right sub-tree and swap with it. After this exchange the node will have one sub-tree at most and then we remove it grounded on some of the above two rules. Here we have to say that it can be done analogical swap, just that we get the left sub-tree and it is the biggest element.

We leave to the reader to check the correctness of these three steps, as a little exercise.

Now, let's see a **sample removal in action**. Again we will use our ordered tree, which we have displayed at the beginning of this point. For example, let's remove the element with key 11.

The node **11 has two sub-trees** and according to our algorithm, it must be exchanged with the smallest element from the right sub-tree, i.e. with 13. After the exchange, we can remove 11 (it is a leaf). Here is the final result:

Below is the sample code, which implements the described algorithm:

```
/// <summary>Removes an element from the tree if exists
/// </summary>
/// <param name="value">the value to be deleted</param>
public void Remove(T value)
{
   BinaryTreeNode<T> nodeToDelete = Find(value);
   if (nodeToDelete != null)
   {
      Remove(nodeToDelete);
   }
}

private void Remove(BinaryTreeNode<T> node)
{
   // Case 3: If the node has two children.
   // Note that if we get here at the end
   // the node will be with at most one child
   if (node.leftChild != null && node.rightChild != null)
   {
      BinaryTreeNode<T> replacement = node.rightChild;
      while (replacement.leftChild != null)
      {
         replacement = replacement.leftChild;
      }
      node.value = replacement.value;
      node = replacement;
   }

   // Case 1 and 2: If the node has at most one child
   BinaryTreeNode<T> theChild = node.leftChild != null ?
         node.leftChild : node.rightChild;

   // If the element to be deleted has one child
   if (theChild != null)
   {
      theChild.parent = node.parent;

      // Handle the case when the element is the root
      if (node.parent == null)
      {
         root = theChild;
      }
      else
      {
```

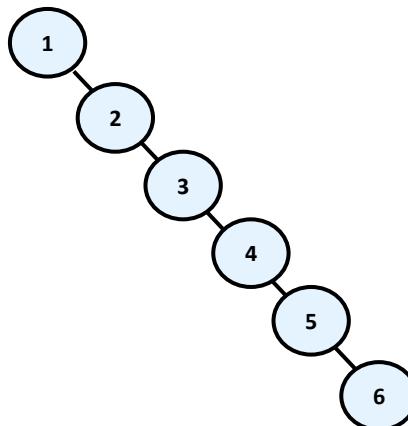
```
        // Replace the element with its child sub-tree
        if (node.parent.leftChild == node)
        {
          node.parent.leftChild = theChild;
        }
        else
        {
          node.parent.rightChild = theChild;
        }
      }
    }
    else
    {
      // Handle the case when the element is the root
      if (node.parent == null)
      {
        root = null;
      }
      else
      {
        // Remove the element - it is a leaf
        if (node.parent.leftChild == node)
        {
          node.parent.leftChild = null;
        }
        else
        {
          node.parent.rightChild = null;
        }
      }
    }
  }
```

We add also a **DFS traversal** method to enable printing the values stored in the tree in ascending order (in-order):

```
/// <summary>Traverses and prints the tree</summary>
public void PrintTreeDFS()
{
  PrintTreeDFS(this.root);
  Console.WriteLine();
}

/// <summary>Traverses and prints the ordered binary search tree
```

```
/// tree starting from given root node.</summary>
/// <param name="node">the starting node</param>
private void PrintTreeDFS(BinaryTreeNode<T> node)
{
  if (node != null)
  {
    PrintTreeDFS(node.leftChild);
    Console.Write(node.value + " ");
    PrintTreeDFS(node.rightChild);
  }
}
```

Finally we demonstrate our **ordered binary search tree** in action:

```
class BinarySearchTreeExample
{
  static void Main()
  {
    BinarySearchTree<string> tree =
      new BinarySearchTree<string>();
    tree.Insert("Telerik");
    tree.Insert("Google");
    tree.Insert("Microsoft");
    tree.PrintTreeDFS(); // Google Microsoft Telerik
    Console.WriteLine(tree.Contains("Telerik")); // True
    Console.WriteLine(tree.Contains("IBM")); // False
    tree.Remove("Telerik");
    Console.WriteLine(tree.Contains("Telerik")); // False
    tree.PrintTreeDFS(); // Google Microsoft
  }
}
```

Note that when we print our **binary search tree**, it is always **sorted** in ascending order (in our case in alphabetical order). Thus in our example the binary search tree of strings behaves like a **set of strings** (we will explain the "Set" data structure in the chapter "Dictionaries, Hash Tables and Sets").

It is important to know that our class **BinarySearchTree<T>** implements a binary search tree, **but not balanced** / **self-balancing** binary search tree. Although it works correctly, its performance can be poor in certain circumstances, like we shall explain in the next section. Balanced trees are more complex concept and use more complex algorithm which guarantees their balanced depth. Let's take a look at them.

Balanced Trees

As we have seen above, the **ordered binary trees** are a very comfortable structure to search within. Defined in this way, the operations for creating and deleting the tree have a hidden flaw: **they don't balance the tree** and its depth could become very big.

Think a bit what will happen if we sequentially include the elements: 1, 2, 3, 4, 5, 6? The ordered binary tree will look like this:

In this case, the **binary tree degenerates into a linked list**. Because of this the searching in this tree is going to be much slower (with **N** steps, not with **log(N)**), as to check whether an item is inside, in the worst case we will have to go through all elements.

We will briefly mention the existence of data structures, which save the logarithmic behavior of the operations adding, searching and removing an element in the common case. We will introduce to you the following definitions before we go on to explain how they are achieved:

Balanced binary tree – a binary tree in which **no leaf is at "much greater" depth than any other leaf**. The definition of "much greater" is rough depends on the specific balancing scheme.

Perfectly balanced binary tree – binary tree in which the difference in **the left and right tree nodes' count** of any node is at most one.

Without going in details we will mention that when given **binary search tree is balanced**, even not perfectly balanced, then the operations of **adding**, **searching** and **removing** an element in it will run in approximately a **logarithmic number of steps** even in the worst case. To avoid imbalance in the tree to search, apply operations that rearrange some elements of the tree when adding or removing an item from it. These operations are called **rotations** in most of the cases. The type of rotation should be further specified and depends on the implementation of the specific data structure. As examples for structures like these we can give **Red-Black tree**, **AVL-tree**, **AA-tree**, **Splay-tree** and others.

Balanced search trees allow quickly (in general case for approximately **log(n)** number of steps) to perform the operations like **searching**, **adding** and **deleting** of elements. This is due to two main reasons:

- Balanced search trees keep their elements **ordered internally**.
- Balanced search trees keep themselves **balanced**, i.e. their depth is always in order of **log(n)**.

Due to their importance in computer science we will talk about **balanced search trees** and their standard implementations in .NET Framework many times when we discuss data structures and their performance in this chapter and in the next few chapters.

Balanced search trees can be binary or non-binary.

Balanced binary search trees have multiple implementations like **Red-Black Trees**, **AA Trees** and **AVL Trees**. All of them are ordered, balanced and binary, so they perform insert / search / delete operations very fast.

Non-binary balanced search trees also have multiple implementations with different special properties. Examples are **B-Trees**, **B+ Trees** and **Interval Trees**. All of them are ordered, balanced, but not binary. Their nodes can typically hold more than one key and can have more than two child nodes. These trees also perform operations like insert / search / delete very fast.

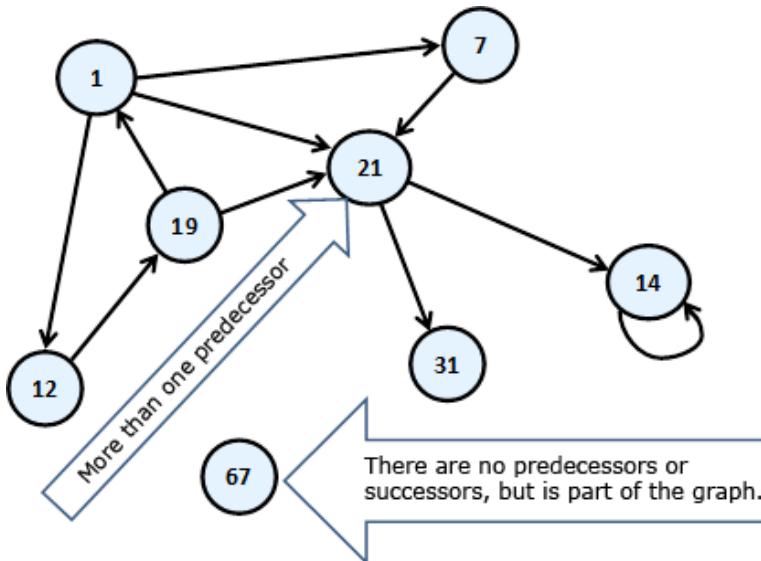
For a more detailed examination of these and other structures we recommend the reader to look closely at literature about algorithms and data structures.

The Hidden Class TreeSet<T> in .NET Framework

Once we have seen ordered binary trees and seen what their advantage is comes the time to show and what C# has ready for us concerning them. Perhaps each of you secretly hoped that he / she will **never have to implement a balanced ordered binary search tree**, because it looks quite complicated.

So far we have looked at what balanced trees are to get an idea about them. When you need to use them, you can always count on getting them from somewhere already implemented. In the standard libraries of the .NET Framework there are ready implementations of balanced trees, but also on the Internet you can find a lot of external libraries.

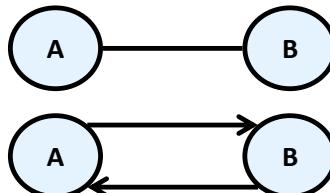
In the namespace **System.Collections.Generic** a class **TreeSet<T>** exists, which is an **implementation of a red-black tree**. This, as we know, means that adding, searching and deleting items in the tree will be made with logarithmic complexity (i.e. if we have one million items operation will be performed for about 20 steps). The bad news is that this class is **internal** and it is visible only in this library. Fortunately, this class is used internally by a class, which is publicly available – **SortedDictionary<T>**. More info about the **SortedDictionary<T>** class you can find in the section "Sets" of chapter "Dictionaries, Hash-Tables and Sets".

Graphs

The **graphs** are very useful and fairly common data structures. They are used to describe a wide variety of **relationships between objects** and in practice can be related to almost everything. As we will see later, trees are a subset of the graphs and also lists are special cases of trees and thus of graphs, i.e. the graphs represent a generalized structure that allows modeling of very large set of real-world situations.

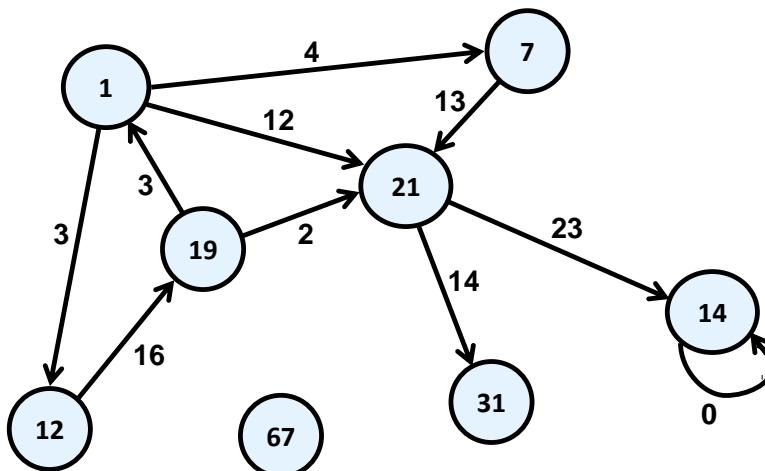
Frequent use of graphs in practice has led to extensive research in "**graph theory**", in which there is a large number of known problems for graphs and for most of them there are well-known solutions.

Graphs – Basic Concepts

In this section we will introduce some of the important concepts and definitions. Some of them are similar to those introduced about the tree data structure, but as we shall see, there are very serious differences, because **trees are just special cases of graphs**.

Let's consider the following **sample graph** (which we would later call a **finite** and **oriented**). Again, like with trees, we have numbered the graph, as it is easier to talk about any of them specifically:

The circles of this scheme we will call **vertices (nodes)** and the arrows connecting them we will call **directed edges**. The vertex of which the arrow comes out we will call **predecessor** of that the arrow points. For example "19" is a predecessor of "1". In this case, "1" is a **successor** of "19". Unlike the structure tree, here each vertex can have more than one predecessor. Like "21", it has three – "19", "1" and "7". If two of the vertices are connected with edge, then we say these two vertices are **adjacent** through this edge.

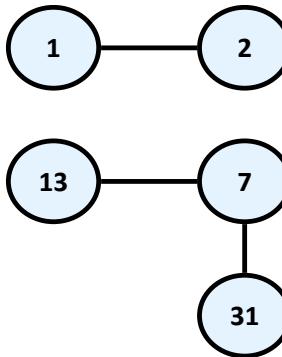
Next follows the definition of **finite directed graph**.

Finite directed graph is called the couple **(V, E)**, in which **V** is a finite set of **vertices** and **E** is a finite set of directed **edges**. Each edge ***e*** that belongs to **E** is an ordered couple of vertices ***u*** and ***v*** or ***e = (u, v)***, which are defining it in a unique way.

For better understanding of this definition we are strongly recommending to the reader to think of the vertices as they are cities, and the directed edges as one-way roads. That way, if one of the vertices is Sofia and the other is Paris, the one-way path (edge) will be called Sofia – Paris. In fact this is one of the classic examples for the use of the graphs – in tasks with paths.

If instead of arrows, the vertices are connected with segments, then the segments will be called **undirected edges**, and the graph – **undirected**. Practically we can imagine that an undirected edge from vertex A to vertex B is two-way edge and equivalent to two opposite directed edges between the same two vertices:

Two vertices connected with an edge are called **neighbors** (adjacent).

For the edges a **weight function** can be assigned, that associates each edge to a real number. These numbers we will call **weights (costs)**. For examples of the weights we can mention some distance between neighboring cities, or the length of the directed connections between two neighboring cities, or the crossing function of a pipe, etc. A graph that has weights on the edges is called **weighted**. Here is how it is illustrated a weighted graph.

Path in a graph is a sequence of vertices v_1, v_2, …, v_n,, such as there is an edge from v_i to v_{i+1} for every i from 1 to n-1. In our example path is the sequence "1", "12", "19", "21". "7", "21" and "1" is not a path because there is no edge starting from "21" and ending in "1".

Length of path is the number of edges connecting vertices in the sequence of the vertices in the path. This number is equal to the number of vertices in the path minus one. The length of our example for path "1", "12", "19", "21" is three.

Cost of path in a weighted graph, we call the sum of the weights (costs) of the edges involved in the path. In real life the road from Sofia to Madrid, for example, is equal to the length of the road from Sofia to Paris plus the length of the road from Madrid to Paris. In our example, the length of the path "1", "12", "19" and "21" is equal to 3 + 16 + 2 = 21.

Loop is a path in which the initial and the final vertex of the path match. Example of vertices forming loop are "1", "12" and "19". In the same time "1", "7" and "21" do not form a loop.

Looping edge we will call an edge, which starts and ends in the same vertex. In our example the vertex "14" is looped.

A **connected undirected graph** we call an undirected graph in which there is a path from each node to each other. For example, the following graph is **not connected** because there is no path from "1" to "7".

So we already have enough knowledge to define the concept tree in other way, as a special kind of graph:

Tree – undirected connected graph without loops.

As a small exercise we let the reader show why all definitions of tree we gave in this chapter are equivalent.

Graphs – Presentations

There are a lot of different ways to present a graph in the computer programming. Different representations have different properties and what exactly should be selected depends on the particular algorithm that we want to apply. In other words – we present the graph in a way, so that the

operations that our algorithm does on it to be as fast as possible. Without falling into greater details we will set out some of the most common representations of graphs.

- **List of successors** – in this representation for each vertex **v** a list of successor vertices is kept (like the tree's child nodes). Here again, if the graph is weighted, then to each element of the list of successors an additional field is added indicating the weight of the edge to it.
- **Adjacency matrix** – the graph is represented as a square matrix **g[N][N]**, where if there is an edge from $\mathbf{v_i}$ to $\mathbf{v_j}$, then the position **g[i][j]** is contains the value 1. If such an edge does not exist, the field **g[i][j]** is contains the value 0. If the graph is weighted, in the position **g[i][j]** we record weight of the edge, and matrix is called a **matrix of weights**. If between two nodes in this matrix there is no edge, then it is recorded a special value meaning infinity. If the graph is undirected, the adjacency matrix will be symmetrical.
- **List of the edges** – it is represented through the list of ordered pairs (v_i, v_j), where there is an edge from $\mathbf{v_i}$ to $\mathbf{v_j}$. If the graph is weighted, instead ordered pair we have ordered triple, and its third element shows what the weight of the edge is.
- **Matrix of incidence between vertices and edges** – in this case, again we are using a matrix but with dimensions **g[M][N]**, where **N** is the number of vertices, and **M** is the number of edges. Each column represents one edge, and each row a vertex. Then the column corresponding to the edge $\mathbf{(v_i, v_j)}$ will contain 1 only at position **i** and position **j**, and other items in this column will contain 0. If the edge is a loop, i.e. is $\mathbf{(v_i, v_i)}$, then on position **i** we record 2. If the graph we want to represent is oriented and we want to introduce edge from $\mathbf{v_i}$ to $\mathbf{v_j}$, then to position **i** we write 1 and to the position **j** we write -1.

The most commonly used representation of graphs is the **list of successors**.

Graphs – Basic Operations

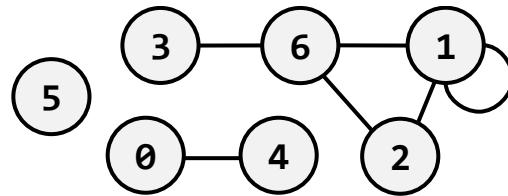
The basic operations in a graph are:

- Creating a graph
- Adding / removing a vertex / edge
- Check whether an edge exists between two vertices
- Finding the successors of given vertex

We will offer a **sample implementation** of the **graph representation with a list of successors** and we will show how to perform most of the operations. This kind of implementation is good when the most often operation we need is to get the list of all successors (child nodes) for a certain vertex. This **graph representation** needs a memory of order N + M where N is the number of vertices and M is the number of edges in the graph.

In essence the vertices are numbered from **0** to **N-1** and our **Graph** class holds for each vertex a list of the numbers of all its child vertices. It does not work with the nodes, but with their numbers in the range [0…N-1]. Let's explore the source code of our sample graph:

```
using System;
using System.Collections.Generic;

/// <summary>Represents a directed unweighted graph structure
/// </summary>
public class Graph
{
   // Contains the child nodes for each vertex of the graph
   // assuming that the vertices are numbered 0 ... Size-1
   private List<int>[] childNodes;

   /// <summary>Constructs an empty graph of given size</summary>
   /// <param name="size">number of vertices</param>
   public Graph(int size)
   {
      this.childNodes = new List<int>[size];
      for (int i = 0; i < size; i++)
      {
         // Assing an empty list of adjacents for each vertex
         this.childNodes[i] = new List<int>();
      }
   }

   /// <summary>Constructs a graph by given list of
   /// child nodes (successors) for each vertex</summary>
   /// <param name="childNodes">children for each node</param>
   public Graph(List<int>[] childNodes)
   {
      this.childNodes = childNodes;
   }

   /// <summary>
   /// Returns the size of the graph (number of vertices)
   /// </summary>
   public int Size
   {
      get { return this.childNodes.Length; }
   }

   /// <summary>Adds new edge from u to v</summary>
```

```
   /// <param name="u">the starting vertex</param>
   /// <param name="v">the ending vertex</param>
   public void AddEdge(int u, int v)
   {
      childNodes[u].Add(v);
   }

   /// <summary>Removes the edge from u to v if such exists
   /// </summary>
   /// <param name="u">the starting vertex</param>
   /// <param name="v">the ending vertex</param>
   public void RemoveEdge(int u, int v)
   {
      childNodes[u].Remove(v);
   }

   /// <summary>
   /// Checks whether there is an edge between vertex u and v
   /// </summary>
   /// <param name="u">the starting vertex</param>
   /// <param name="v">the ending vertex</param>
   /// <returns>true if there is an edge between
   /// vertex u and vertex v</returns>
   public bool HasEdge(int u, int v)
   {
      bool hasEdge = childNodes[u].Contains(v);
      return hasEdge;
   }

   /// <summary>Returns the successors of a given vertex
   /// </summary>
   /// <param name="v">the vertex</param>
   /// <returns>list of all successors of vertex v</returns>
   public IList<int> GetSuccessors(int v)
   {
      return childNodes[v];
   }
}
```

To illustrate how our **graph data structure** works, we will create small program that creates a graph and **traverses it by the DFS algorithm**. To play a bit with graphs, the goal of our graph traversal algorithm will be to count how many **connected components** the graph has.

By definition in **undirected graph** if a **path exists** between two nodes, they belong to the **same connected component** and if **no path exists** between

two nodes, they belong to **different connected components**. For example consider the following undirected graph:

It has 3 connected components: **{0, 4}**, **{1, 2, 6, 3}** and **{5}**.

The code below **creates a graph** corresponding to the figure above and by DFS traversal **finds all its connected components**. This is straightforward: pass through all vertices and once unvisited vertex is found, all connected to it vertices (directly or indirectly via some a path) are found by DFS traversal, each of them is printed and marked as visited. Below is the code:

```
class GraphComponents
{
   static Graph graph = new Graph(new List<int>[] {
      new List<int>() {4},        // successors of vertice 0
      new List<int>() {1, 2, 6}, // successors of vertice 1
      new List<int>() {1, 6},     // successors of vertice 2
      new List<int>() {6},        // successors of vertice 3
      new List<int>() {0},        // successors of vertice 4
      new List<int>() {},         // successors of vertice 5
      new List<int>() {1, 2, 3}  // successors of vertice 6
   });

   static bool[] visited = new bool[graph.Size];

   static void TraverseDFS(int v)
   {
      if (!visited[v])
      {
         Console.Write(v + " ");
         visited[v] = true;
         foreach (int child in graph.GetSuccessors(v))
         {
            TraverseDFS(child);
         }
      }
   }

   static void Main()
   {
```

```
        Console.WriteLine("Connected graph components: ");
        for (int v = 0; v < graph.Size; v++)
        {
            if (!visited[v])
            {
                TraverseDFS(v);
                Console.WriteLine();
            }
        }
    }
}
```

If we run the above code, we will get the following output (the connected components of our sample graph shown above):

```
Connected graph components:
0 4
1 2 6 3
5
```

Common Graph Applications

Graphs are used to model many situations of reality, and tasks on graphs model multiple real problems that often need to be resolved. We will give just a few examples:

- **Map of a city** can be modeled by a **weighted oriented graph**. On each street, edge is compared with a length, corresponding to the length of the street, and direction – the direction of movement. If the street is a two-way, it can be compared to two edges in both directions. At each intersection there is a node. In such a model there are natural tasks such as searching for the **shortest path between two intersections**, checking whether there is a road between two intersections, checking for a loop (if we can turn and go back to the starting position) searching for a path with a minimum number of turns, etc.

- **Computer network** can be modeled by an **undirected graph**, whose vertices correspond to the computers in the network, and the edges correspond to the communication channels between the computers. To the edges different numbers can be compared, such as channel capacity or speed of the exchange, etc. Typical tasks for such models of a network are **checking for connectivity** between two computers, checking for **double-connectivity** between two points (existence of double-secured channel, which remains active after the failure of any computer), finding a **minimal spanning tree** (MST), etc. In particular, the **Internet can be modeled as a graph**, in which are solved

problems for routing packets, which are modeled as classical graph problems.

- **The river system** in a given region can be modeled by a **weighted directed graph**, where each river is composed of one or more edges, and each node represents the place where two or more rivers flow into another one. On the edges can be set values, related to the amount of water that goes through them. Naturally with this model there are tasks such as calculating the volume of water, passing through each vertex and anticipate of possible flood in increasing quantities.

You can see that the **graphs** can be used to solve many **real-world problems**. Hundreds of books and research papers are written about graphs, graph theory and graph algorithms. There are dozens of classic tasks for graphs, for which there are known solutions or it is known that there is no efficient solution. The scope of this chapter does not allow mentioning all of them, but we hope that through the short presentation we have awaken your interest in **graphs, graph algorithms and their applications** and spur you to take enough time to solve the tasks about graphs in the exercises.

Exercises

1. Write a program that **finds the number of occurrences of a number in a tree** of numbers.
2. Write a program that displays the roots of those **sub-trees** of a tree, which have exactly **k** nodes, where **k** is an integer.
3. Write a program that finds the **number of leaves** and **number of internal vertices** of a tree.
4. Write a program that finds in a binary tree of numbers the **sum of the vertices of each level** of the tree.
5. Write a program that finds and **prints all vertices of a binary tree**, which have for only leaves successors.
6. Write a program that **checks whether a binary tree is perfectly balanced**.
7. Let’s have as given a **graph G(V, E)** and two of its vertices **x** and **y**. Write a program that **finds the shortest path** between two vertices measured in number of vertices staying on the path.
8. Let’s have as given a graph **G(V, E)**. Write a program **that checks whether the graph is cyclic**.
9. Implement a **recursive traversal in depth** in an undirected graph and a program to test it.
10. Write **breadth first search (BFS)**, based on a queue, to traverse a directed graph.

11. Write a program that **searches the directory `C:\Windows\`** and all its subdirectories recursively and prints all the files which have extension **`*.exe`**.

12. Define classes **`File {string name, int size}`** and **`Folder {string name, File[] files, Folder[] childFolders}`**. Using these classes, build a tree that contains all files and directories on your hard disk, starting from **`C:\Windows\`**. Write a method that calculates the sum of the sizes of files in a sub-tree and a program that tests this method. To crawl the directories use **recursively crawl depth (DFS)**.

13. * Write a program that **finds all loops in a directed graph**.

14. Let's have as given a graph **G (V, E)**. Write a program that **finds all connected components** of the graph, i.e. finds all maximal connected sub-graphs. A maximal connected sub-graph of **G** is a connected graph such that no other connected sub-graphs of **G**, contains it.

15. Suppose we are given a **weighted oriented graph G (V, E)**, in which the weights on the side are nonnegative numbers. Write a program that by a given vertex **x** from the graph finds the **shortest paths from it to all other vertical**.

16. We have **N tasks to be performed successively**. We are given a list of pairs of tasks for which the second is dependent on the outcome of the first and should be executed after it. Write a program that **arranges tasks in such a way** that each task is be performed after all the tasks which it **depends** on have been completed. If no such order exists print an appropriate message.

    Example: **`{1, 2}, {2, 5}, {2, 4}, {3, 1}` → `3, 1, 2, 5, 4`**

17. An **Eulerian cycle** in a graph is called a loop that starts from a vertex, passes exactly once through all edges in the graph returns to the starting vertex. Vertices can be visited repeatedly. Write a program that by a given graph, **finds whether the graph has an Euler loop**.

18. A **Hamiltonian cycle** in a graph is a cycle containing every vertex in the graph exactly once. Write a program, which by given weighted oriented graph G (V, E), **finds Hamiltonian loop with a minimum length**, if such exists.

Solutions and Guidelines

1. **Traverse the tree recursively** in depth (using DFS) and count the occurrences of the given number.

2. **Traverse the tree recursively** in depth (using DFS) and check for each node the given condition. For each node the number of nodes in its subtree is: 1 + the sum of the nodes of each of its child subtrees.

3. You can solve the problem by **traversing the tree in depth** recursively.

4. Use **traversing in depth** or **breadth** and when shifting from one node to another keep its **level** (**depth**). Knowing the levels of the nodes at each step, the wanted amount can be easily calculated.

5. You can solve the problem by **recursively traversing the tree** in depth and by checking the given condition.

6. By **recursive traversal in depth** (**DFS**) for every node of the tree calculate the depths of its left and right sub-trees. Then check immediately whether the condition of the definition for perfectly balanced tree is executed (check the difference between the left and right sub-tree's depths).

7. Use the algorithm of **traversing in breadth** (**BFS**) as a base. In the queue put every node always **along with its predecessor**. This will help you to restore the path between the nodes (in reverse order).

8. Use **traversing in depth or in breadth**. Mark every node, if already visited. If at any time you reach to a node, which has **already been visited**, then you have found loop.

   Think about how you can **find and print the loop itself**. Here is an idea: while traversing every node **keep its predecessor**. If at any moment you reach a node that has already been visited, you should have a path to the initial node. The current path in the recursion stack is also a path to the wanted node. So at some point we have two different paths from one node to the initial node. **By merging the two paths** you can easily find the loop.

9. Use the **DFS** algorithm. Testing can be done with few example graphs.

10. Use the **BFS** algorithm. Instead of putting the vertices of the graph in the queue, put their numbers (0 … N-1). This will simplify the algorithm.

11. Use **traversing in depth** and **`System.IO.Directory`** class.

12. Use the example of the tree data structure given in this chapter. Each directory from the tree should two arrays (or lists) of descendants: **subdirectories** and **files**.

13. Use the solution of **problem 8**, but modify it so it does not stop when it finds a loop, but continues. For each loop you should check if you have already found it. This problem is more complex than you may expect!

14. Use the **algorithms for traversing in breadth or depth** as a base.

15. Use the **Dijkstra's algorithm** (find it on the Internet).

16. The requested order is called "**topological sorting** of a directed graph". It can be implemented in two ways:

    For every task **`t`** we should know how many others tasks **`P(t)`** it depends on. We find task t_0, which is independent, i.e. $P(t_0)=0$ and we execute it. We reduce **`P(t)`** for every task, which depends from task t_0. Again we look for a task, which is independent and we execute it. We repeat until

the tasks end or until we find a moment when there is no task t_k having **P(t_k)=0**. In the last case no solution exists due to a cyclic dependency.

We can solve the task with **traversing the graph in depth** and printing every node just before leaving it. That means that at any time of printing of a task, all the tasks that depend on it should have already been printed. The topological sorting will be produced in **reversed order**.

17. The graph must be **connected and the degree of each of its nodes must be even** in order an **Eulerian cycle** in a graph to exits (can you prove this?). With series of DFS traversals you can find **cycles in the graph** and to remove the edges involved in them. Finally, by **joining the cycles** you will get the Eulerian cycle. See more about Eulerian paths and cycles at http://en.wikipedia.org/wiki/Eulerian_path.

18. If you write a **true solution** of the problem, check whether it works for a graph with 200 nodes. Do not try to solve the problem so it could work with a large number of nodes! If someone manages to solve it for large numbers of nodes, he will remain permanently in history! See also the Wikipedia article http://en.wikipedia.org/wiki/Hamiltonian_path_problem. You might try some **recursive algorithm for generating all paths** but accept that it will be slow. Techniques like backtracking and branch and bound could help a bit but generally this problem is NP-complete and thus **no efficient solution is known to exist** for it.

Chapter 18. Dictionaries, Hash-Tables and Sets

In This Chapter

In this chapter we will analyze more complex data structures like **dictionaries and sets**, and their **implementations with hash-tables and balanced trees**. We will explain in more details what **hashing** and **hash-tables** mean and why they are such an important part of programming. We will discuss the concept of "**collisions**" and how they might happen when implementing hash-tables. Also we will offer you different types of approaches for solving this type of issues. We will look at the abstract data structure **set** and explain how it can be implemented with the ADTs **dictionary** and **balanced search tree**. Also we will provide you with examples that illustrate the behavior of these data structures with real world examples.

Dictionary Data Structure

In the last few chapters we got familiar with some classic and very important data structures – arrays, lists, trees and graphs. In this chapter we will get familiar with the so called "**dictionaries"**, which are extremely useful and widely used in the programming.

The dictionaries are also known as **associative arrays** or **maps**. In this book we are going to use the terminology "dictionary". Every element in the dictionary has a **key** and an **associated value** for this key. Both the key and the value represent a pair. The analogy with the real world dictionary comes from the fact, that in every dictionary, for every for word (**key),** we also have a description related to this word (**value**).

	As well as the data (values), that the dictionary holds, there is also a key that is used for searching and finding the required values. The elements of the dictionary are represented by pairs (key, value), where the key is used for searching.

Dictionary Data Structure – Example

We are going to illustrate what exactly the data structure dictionary means using an everyday, real world example.

When you go to a theatre, opera or a concert, there is usually a place where you can leave your outdoor clothing. The employee than takes your jacket and gives you a number. When the event is over, on your way out, you give them back the same number. The employee uses this number to search and find your jacket to give back to you.

Thanks to this example we can see that the idea for using a **key** (the number that the employee gives you) to store a **value** (your jacket), and later having the option to access it, is not so abstract. Actually this is a method that is often widely used not only in programming, but also in many other practical areas.

When using the **ADT dictionary**, the key may not just be a number, but any other type of object. In the case, when we have a key (number), we could implement this type of structure as a regular array. In this scenario the **set of keys** is already known – these are the numbers from **0** to **n**, where **n** represents the size of the array (when **n** is within the allowed limits). The idea of the dictionaries is to provide us with more flexibility regarding the set of the keys.

When using dictionaries, the set of keys usually is a randomly chosen set of values like real numbers or strings. The only restriction is that we can distinguish one key from the other. Later we will take a look at some additional requirements for the keys that are needed for the different kinds of implementations.

For every **key** in the dictionary, there is a **corresponding value**. One key can hold only one value. The aggregation of all the **pairs (key, value)** represents the dictionary.

Here is the first example for using a dictionary in .NET:

```
IDictionary<string, double> studentMarks =
  new dictionary<string, double>();

studentMarks["Paul"] = 3.00;
Console.WriteLine("Paul 's mark: {0:0.00}",
  studentMarks["Paul"]);
```

Later in this chapter, we will find out the result from the execution of the example above.

The Abstract Data Structure "Dictionary" (Associative Array, Map)

In programming the **abstract data structure "dictionary"** is represented by many aggregated pairs (key, value) along with predefined methods for accessing the values by a given key. Alternatively this data structure can also be called a "**map**" or "**associative array**".

Described below are the required operations, defined by this data structure:

- **void Add(K key, V value)** – adds given key-value pair in the dictionary. With most implementations of this class in .NET, when adding a key that already exists, an **exception is thrown**.
- **V Get(K key)** – returns the value by the specified key. If there is no pair with this key, the method returns **null** or throws an exception depending on the specific dictionary implementation.
- **bool Remove(key)** – removes the value, associated with the specified key and returns a Boolean value, indicating if the operation was successful.

Here are some additional methods, which are supported by the ADT.

- **bool Contains(key)** – returns **true** if the dictionary has a pair with the selected key
- **int Count** – returns the number of elements (key value pairs) in the dictionary

Other operations that are usually supported are: extracting all of the keys, values or key value pairs and importing them into another structure (array, list). This way they can easily be traversed using a loop.

For the comfort of .NET developers, the IDictionary<K, V> interface holds an indexing property V this[K] { get; set; }, which is usually implemented by calling the methods V Get(K), Add(K, V).

Bear in mind that the access method (accessor) get of the property V this[K] of the class Dictionary<K, V> in .NET throws an exception if the given key K does not exist in the dictionary. In order to access the value of a certain key, without having to worry about exceptions, use the method bool TryGetValue(K key, out V value).

The Interface IDictionary<K, V>

In .NET there is a standard interface **IDictionary<K, V>** where **K** defines the type of the **key**, and **V** type of the **value**. It defines all of the basic operations that the dictionaries should implement. **IDictionary<K, V>** corresponds to the abstract data structure "dictionary" and defines the operations, mentioned above, but without supplying an actual implementation of them. This interface is defined in assembly **mscorelib**, **namespace System.Collections.Generic**.

In .NET **interfaces** represent **specifications of methods** for a certain class. They define methods without implementation, which should be implemented by the classes that inherit them. How the interfaces and inheritance work we will discuss in more details in the chapter "Principles of the Object-Oriented Programming". For the moment all you need to know is that interfaces define

which methods and fields should be implemented in the classes that inherit the interface.

In this chapter we will take a look at the two most popular dictionary implementations – with a **balanced tree** and a **hash-table**. It's extremely important for you to know how they differ from one another, and which are the main principles related to them. Otherwise you risk using them improperly and inefficiently.

In .NET Framework there are two major implementations of the interface **IDictionary<K, V> – Dictionary<K, V>** and **SortedDictionary<K, V>**. **SortedDictionary** is an implementation by a balanced (red-black) tree, and **Dictionary** – by a hash-table.

Except for IDictionary<K, V> in .NET there is one more interface – IDictionary, along with the classes implementing it: Hashtable, ListDictionary and HybridDictionary. They are heritage from the first version of .NET. These classes need to be used only on special occasions. Much more preferable is the use of Dictionary<K, V> or SortedDictionary<K, V>.

In this and the next chapter we will analyze when the different implementations of dictionaries are used.

Implementation of Dictionary with Red-Black Tree

Because the **implementation of a dictionary with a balanced tree** is very extensive and complex task, we will not examine it in source code. Instead we will analyze the class **SortedDictionary<K, V>**, that comes with the standard .NET library. We strongly recommend the curious readers to look at the decompiled code of the **SortedDictionary** class using some of the decompilation tools mentioned in the chapter "Introduction to Programming" like JustDecompile.

As we mentioned in the previous chapter, a **red-black tree is an ordered binary balanced search tree**, that's used for searching. This is why one of the important requirements for the set of keys used by **SortedDictionary<K, V>** is **comparability**. This means that, if we have two keys, either one of them should be bigger, or they should be equal. The keys used in **SortedDictionary<K, V>** should implement **IComparable<K>**.

The usage of the **binary search tree** gives us a great advantage: the **keys in the dictionary are stored ordered**. Thanks to this feature, if we need the data ordered by keys, we don't need to perform any additional sorting. Actually, this is the only advantage of this dictionary implementation compared to the **hash-table**.

A thing that should be mentioned is that keeping the keys ordered comes with its price. Searching for the elements using in an **ordered balanced tree** is slower (typically takes **log(n)** steps) than using **a hash-table** (typical takes

fixed number of steps). Because of this, if there is no requirement for the keys to be ordered, it's better to use **Dictionary<K, V>**.

	Use a balanced tree dictionary only when you need your pairs (key, value) to be ordered by key. Bear in mind that the balanced tree comes with the complexity of the algorithm log(n)**, for searching, adding and deleting elements. Compared to this, the complexity used in hash-table may reach a linear value.**

The Class SortedDictionary<K, V>

The class **SortedDictionary<K, V>** is a dictionary implementation, which uses a **red-black tree**. This class implements all the standard operations defined in the interface **IDictionary<K, V>**.

Using SortedDictionary Class – Example

Now we will solve a practical problem, where using the class **SortedDictionary** is a good idea. Let's say we have arbitrary text. Our task would be to **find all the different words in the text**, and the number of occurrences of these words. Additionally we should print all the words found in alphabetical order.

For this task using a **dictionary is a really good idea**. We can use the different words in the text for keys, and the value for each key would be the number of occurrences for each word in our text.

The **algorithm for counting the words** is the following: we read the text word by word. For each word we check if it already exists in the dictionary. If the answer is no, we add a new element in the dictionary with a value of 1. If the answer is yes – we increase the old value of the element by one, so as to count the last occurrence.

The elements of the ordered dictionary **SortedDictionary<string, int>** will be ordered by their key during the iteration process. This way we met the additional requirement for the words to be ordered alphabetically. Below is a **sample implementation** of the described algorithm:

WordCountingWithSortedDictionary.cs

```
using System;
using System.Collections.Generic;

class WordCountingWithSortedDictionary
{
  private static readonly string Text =
    "Mary had a little lamb " +
    "little Lamb, little Lamb, " +
```

```
        "Mary had a Little lamb, " +
        "whose fleece were white as snow.";

    static void Main()
    {
        IDictionary<String, int> wordOccurrenceMap =
            GetWordOccurrenceMap(Text);
        PrintWordOccurrenceCount(wordOccurrenceMap);
    }

    private static IDictionary<string, int> GetWordOccurrenceMap(
        string text)
    {
        string[] tokens =
            text.Split(' ', '.', ',', '-', '?', '!');

        IDictionary<string, int> words =
            new SortedDictionary<string, int>();

        foreach (string word in tokens)
        {
            if (!string.IsNullOrEmpty(word.Trim()))
            {
                int count;
                if (!words.TryGetValue(word, out count))
                {
                    count = 0;
                }
                words[word] = count + 1;
            }
        }
        return words;
    }

    private static void PrintWordOccurrenceCount(
        IDictionary<string, int> wordOccurenceMap)
    {
        foreach (var wordEntry in wordOccurenceMap)
        {
            Console.WriteLine(
                "Word '{0}' occurs {1} time(s) in the text",
                 wordEntry.Key, wordEntry.Value);
        }
    }
```

```
}
```

The output from executing this code is the following:

```
Word 'a' occurs 2 time(s) in the text
Word 'as' occurs 1 time(s) in the text
Word 'fleece' occurs 1 time(s) in the text
Word 'had' occurs 2 time(s) in the text
Word 'lamb' occurs 2 time(s) in the text
Word 'Lamb' occurs 2 time(s) in the text
Word 'little' occurs 3 time(s) in the text
Word 'Little' occurs 1 time(s) in the text
Word 'mary' occurs 2 time(s) in the text
Word 'snow' occurs 1 time(s) in the text
Word 'was' occurs 1 time(s) in the text
Word 'white' occurs 1 time(s) in the text
Word 'whose' occurs 1 time(s) in the text
```

Note that we are counting the words "little" and "lamb" starting with both lowercase and uppercase characters as different.

In this example, we demonstrated for the first time how to traverse a dictionary using the method **PrintWordOccurrenceCount(IDictionary <string, int>)**. We used a **foreach** loop. When iterating through the elements of dictionaries, we need to take into account that the elements of this ADT are ordered pairs (key and value), not just single objects. Because **IDictionary<K, V>** implements the interface **IEnumerable<KeyValuePair <K, V>>**, this means that the **foreach** loop should iterate through objects of type **KeyValuePair<K, V>**. For simplicity we use the **var**-syntax in the **foreach** loop.

IComparable<K> Interface

When using **SortedDictionary<K, V>** the keys are required to be **comparable**. In our example we use objects of type **string**.

The class **string** implements the interface **IComparable**, and the comparison between the elements is done lexicographically. What does that mean? By default the strings in .NET are case sensitive (the compiler distinguishes uppercase from lowercase letters). Words like "Length" and "length" are considered different. This means that words that start with a lowercase letter will be before the ones with an uppercase letter. This definition comes from the implementation of the method **CompareTo(object)**, through which the **string** class implements the interface **IComparable**.

IComparer<T> Interface

What should we do when we are not happy with the default implementation of comparison? For example, what should we do when we want uppercase and lowercase characters to be treated as equal?

One option we have is to transform the word into a capital, or non-capital string, but sometimes the situation is more complicated than that. This is why we will offer another solution, which works for every class that does not implement the **IComparable<T>** interface, or it does implement it, but we want to change its behavior.

For the comparison of objects with an exclusively defined order in **SortedDictionary<K, V>** in .NET, we will use the interface **IComparer<T>**. It defines a comparison function **int Compare(T x, T y)** that is an alternative to the already defined order. Let's take a better look at this interface.

When we create an object of type **SortedDictionary<K, V>** we can pass to its constructor a reference to **IComparer<K>** so that it can use it for the key comparison (key elements should be objects of type **K**).

Here is a sample implementation of **IComparer<K>** that changes the behavior when comparing strings, so that they are **not** distinguished by uppercase and lowercase characters:

```
class CaseInsensitiveComparer : IComparer<string>
{
  public int Compare(string s1, string s2)
  {
    return string.Compare(s1, s2, true);
  }
}
```

Let's use this interface **IComparer<E>** when creating the dictionary:

```
IDictionary<string, int> words =
  new SortedDictionary<string, int>(
    new CaseInsensitiveComparer());
```

After changing this in the code, the result from the program execution will be:

```
Word 'a' occurs 2 time(s) in the text
Word 'as' occurs 1 time(s) in the text
Word 'fleece' occurs 1 time(s) in the text
Word 'had' occurs 2 time(s) in the text
Word 'lamb' occurs 4 time(s) in the text
Word 'little' occurs 4 time(s) in the text
…
```

The first time a word is found, it becomes a key in the dictionary. This is because after calling the **words[word] = count + 1** only the value is changed, and not the key itself.

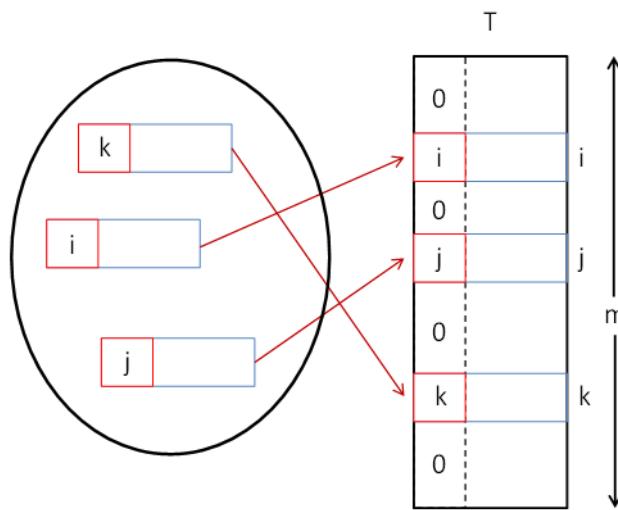
After using **IComparer<E>** we changed the definition for ordering keys in our dictionary. If, for a key, we used a class, defined by us, for example – **Student**, that implements **IComparable<E>**, we would get the same result if we were to alter the method **CompareTo(Student)**. There is also one additional requirement, when implementing **IComparable<K>**:

When two objects are equal (Equals(object) returns true), CompareTo(E) should return 0.

Meeting this requirement would allow us to use the objects of a custom class as keys, just as in the implementation with a balanced tree (**SortedDictionary<K,V>**, constructed without **Comparer**), as well with a hash-table (**Dictionary<K,V>**).

Hash-Tables

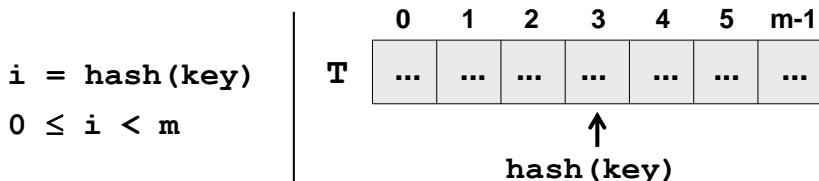
Now let's get familiar with the data structure **hash-table**, which implements the abstract data structure **dictionary** in a **very efficient way**. We well explain in details how hash-tables actually work and why they are so efficient.

Dictionary Implementation with Hash-Table

With a **hash-table implementation**, the time for accessing the elements in the dictionary is theoretically **independent from their count**. This is a very important advantage.

Let's make a comparison between **list** and **hash-table** in the speed of searching. We take a **list** of randomly ordered elements. We want to check if a certain element is in the list. The worst case scenario is to check every element in the list, so as to give an explicit answer to the question "Does this list contain the element or not". It's obvious that the number of checks would depend (linear) of the number of elements.

With **hash-tables**, if we have a key, the number of comparisons that we would need to do to find out if there is a key with this value, is **constant** and it **does not depend on the number of elements**. How exactly we are achieving such efficiency, we will explain in more details below.

What is a Hash-Table?

The data structure **hash-table** is usually implemented internally with an **array**. It consists of **numerated elements** (cells), each either **holding a key-value pair** or is **empty** (**null**). This at first sight, look like as if the elements were randomly placed in the array. At the positions that we don't have an ordered pair, we have an empty element (**null**). The figure below illustrates how a hash-table might look like:

The size of the internal storage array of the hash-table is called **capacity**. The **load factor** is a real number between 0 and 1, which stands for the ratio between the occupied elements and the current capacity. At the figure we have a hash-table with **3** elements and capacity **m**. The load factor for this hash-table would be **3/m**.

When adding or searching for elements, a method for hashing the key (**hash function**) is executed **hash(key)**, that returns a number we call a **hash-code**. When we take the division remainder of this hash-code and the capacity **m** we get a number between **0** and **m-1**:

```
index = hash(key) % m
```

At the figure there is a hash-table **T** with capacity **m** and hash-function **hash(key)**:

This value **hash(k)** gives us the **position** in the array at which we search or add a certain **key-value pair** having this **k**. If the hash-function distributes the keys uniformly, in most cases for every key a different hash value will be assigned. In this way **every cell of the array will have at most one key**. Ultimately we get an extremely fast search and insertion of the elements: just **calculate the hash function and obtain the cell assigned for the key**. Of course it may occur that different keys would have the same hash code. We will examine this special case in more details later.

Use implementation of dictionary based on hash-table, when you need to find values by key with a maximum speed.

The internal table's **capacity is increased** when the number of elements in the hash-table becomes greater or equal to a certain constant called **fill factor** (load factor, the maximal degree of filling). When increasing the capacity (usually doubling it), all of the elements are reordered by the hash code of their keys and their assigned cell is calculated according to the new capacity. The **load factor** is significantly decreased after the reordering. This operation is time-consuming, but it is executed relatively rare, so it will not impact the overall performance of the "add" operation.

Before we go further with the theory of hash-tables, let's review how hash-tables are implemented in C# and .NET Framework.

Class Dictionary <K, V>

The class **`Dictionary<K, V>`** is a standard implementation of a **dictionary based on hash-table** in .NET Framework. Let's take a look at its main features. We will examine a specific example that illustrates the use of this class and its methods.

Class Dictionary<K, V> – Main Operations

Creating a hash-table is done by calling some of the constructors of **`Dictionary<K, V>`**. Through them we can assign an initial value for the capacity and load factor. It's good if we know in advance the expected number of elements, which would be added in our hash-table, so as to set it at the creation of the hash-table. This way we will avoid the unneeded expansions of the hash-table and we will achieve better performance. By default the value of the **initial capacity is 16**, and the **load factor is 0.75**.

Let's review the methods in the class **`Dictionary<K, V>`**:

- **`void Add(K, V)`** adds a new pair (key and a value) to the hash-table. Throws an exception in the case that the key exists. This operation is extremely fast.
- **`bool TryGetValue(K, out V)`** returns an element of type V via the **`out`** parameter for the given key or **`null`**, if there is no such key. The result of this operation will be **`true`** if such an element is found. The operation is very fast, because the algorithm for searching an element by key in the hash-table is with complexity about O(1)
- **`bool Remove(K)`** removes the element with this key. This operation works very fast.
- **`void Clear()`** removes all the elements from the dictionary.
- **`bool ContainsKey(K)`** check if there is an ordered pair with this key in the dictionary. This operation works extremely fast.

- **bool ContainsValue(V)** checks if there is one or more ordered pairs with this value. This operation is slow because it checks every element of the hash-table (like searching in a list).
- **int Count** returns the number of ordered pairs within the dictionary.
- Other operations – extracting all the keys, values or ordered pairs into a structure that could be iterated through using a loop.

Students and Marks – Example

We will illustrate how to use some of the above described operations with an example. We have some students, and every one of them could have only one mark. We want to store the marks in a structure that would allow us to perform a **fast search by the student's name**.

For this task we create a **hash-table** with initial capacity of 6. It will use the student names for keys, and their marks for values. We will add 6 sample students, and then we will check what's happening when we print their data on the console. Here is how the code for this example should look like:

```
using System;
using System.Collections.Generic;

class StudentsExample
{
  static void Main()
  {
    IDictionary<string, double> studentMarks =
      new Dictionary<string, double>(6);

    studentMarks["Alan"] = 3.00;
    studentMarks["Helen"] = 4.50;
    studentMarks["Tom"] = 5.50;
    studentMarks["James"] = 3.50;
    studentMarks["Mary"] = 4.00;
    studentMarks["Nerdy"] = 6.00;

    double marysMark = studentMarks["Mary"];
    Console.WriteLine("Mary's mark: {0:0.00}", marysMark);
      studentMarks.Remove("Mary");

    Console.WriteLine("Mary's mark removed.");

    Console.WriteLine("Is Mary in the dictionary: {0}",
      studentMarks.ContainsKey("Mary") ? "Yes!": "No!");

    Console.WriteLine("Nerdy's mark is {0:0.00}.",
```

```
      studentMarks["Nerdy"]);
    studentMarks["Nerdy"] = 3.25;

    Console.WriteLine(
      "But we all know he deserves no more than {0:0.00}.",
      studentMarks["Nerdy"]);

    double annasMark;
    bool findAnna = studentMarks.TryGetValue("Anna",
      out annasMark);

    Console.WriteLine(
      "Is Anna's mark in the dictionary? {0}",
      findAnna ? "Yes!": "No!");

    studentMarks["Anna"] = 6.00;
    findAnna = studentMarks.TryGetValue("Anna",
      out annasMark);

    Console.WriteLine(
      "Let's try again: {0}. Anna's mark is {1}",
      findAnna ? "Yes!" : "No!", annasMark);

    Console.WriteLine("Students and marks:");

    foreach (KeyValuePair<string, double> studentMark
      in studentMarks)
    {
      Console.WriteLine("{0} has {1:0.00}",
        studentMark.Key, studentMark.Value);
    }

    Console.WriteLine(
      "There are {0} students in the dictionary",
      studentMarks.Count);
    studentMarks.Clear();
    Console.WriteLine("Students dictionary cleared.");
    Console.WriteLine("Is dictionary empty: {0}",
      studentMarks.Count == 0);
  }
}
```

The **output** of the program execution will be:

```
Mary's mark: 4.00
Mary's mark removed.
Is Mary in the dictionary: No!
Nerdy's mark is 6.00.
But we all know he deserves no more than 3.25.
Is Anna's mark in the dictionary? No!
Let's try again: Yes!. Anna's mark is 6
Students and marks:
Alan has 3.00
Helen has 4.50
Tom has 5.50
James has 3.50
Anna has 6.00
Nerdy has 3.25
There are 6 students in the dictionary
Students dictionary cleared.
Is dictionary empty: True
```

We can see that the students are not ordered when printed. This is because in hash-tables (unlike balanced trees) the elements **are not kept sorted**.

Even if the current table capacity is changed while working with it, it is also highly possible that the order of the pairs could be changed as well. We will analyze the reason for this behavior later on.

It is important to remember, that with hash-tables, we cannot rely on the elements being in order. If we need them ordered, we could sort the elements before printing. Another option would be using **`SortedDictionary<K, V>`**.

Hashing and Hash-Functions

Now we will explain in more details the concept of hash-code used earlier. The **hash-code** is a number returned by the **hash-function**, used for the **hashing the key**. This number should be different for every key, or at least there should be a high chance for that.

Hash-Functions

There is the concept of the **perfect hash-function**. One hash-function is called **perfect**, if for example you have N keys, and for each of them the function would add a **different number** in a reasonable interval (for example from 0 to N-1).

Finding such a function in the common case is a very hard, **almost impossible** task. It’s worth to use such functions when using sets of keys with predefined elements or when the set of keys is rarely changed.

In practice there are also other, **not so "perfect" hash-functions**.

Now we will take a look at a few examples for hash-functions, which are used directly with .NET libraries.

The Method GetHashCode() in .NET Framework

Every .NET class has a method called **GetHashCode()** that returns a value of type **int**. This method is inherited by the class **Object**, which is the root member in the hierarchy of .NET classes.

The implementation in the class **Object** of the method **GetHashCode()** **does not guarantee** the unique value of the result. This means that the descendent classes need to ensure that **GetHashCode()** is implemented in order to use it for a key in a hash-table.

Another example for a hash-function that is directly built in .NET is used by the class **int**, **byte** and **short** (integer numbers). In this case the value of the number itself is used for the hash-code. For more complex types like strings all their elements (or at least the first few of them) are involved into calculation of their hash code.

One more complex **example for hash-function** is the implementation of **GetHashCode()** in the class **System.String**:

```
public override unsafe int GetHashCode()
{
  fixed (char* str = ((char*)this))
  {
    char* chPtr = str;
    int num = 352654597;
    int num2 = num;
    int* numPtr = (int*)chPtr;
    for (int i = this.Length; i > 0; i -= 4)
    {
      num = (((num << 5) + num) + (num >> 27)) ^ numPtr[0];
      if (i <= 2)
      {
        break;
      }
      num2 = (((num2 << 5) + num2) + (num2 >> 27)) ^ numPtr[1];
      numPtr += 2;
    }
    return (num + (num2 * 1566083941));
  }
}
```

This implementation is complicated, but what we need to remember is that it tries to guarantee the **uniqueness** of the result: different hash code for different input strings. Note that the complexity of the algorithm for

calculating the hash-code of **`string`** is proportional to **`Length / 4`** or **`O(n)`**, which means that the longer the string is the slower its hash-code would be calculated. Authors of the above code use a small trick (**`unsafe`** code) to directly work with the low-level representation of the string in the memory.

We leave to the reader to take a look at other implementations of the method **`GetHashCode()`** in some of the most commonly used .NET types like **`int`**, **`DateTime`**, **`long`**, **`float`** and **`double`**. This can be done through a decompiler like JustDecompile.

Now let's answer the question of how to implement ourselves this **hash function** for our classes. We already explained that leaving the implementation that is already built in the class **`object`**, is not an acceptable solution. Another very simple implementation is that we always return a fixed constant, for example:

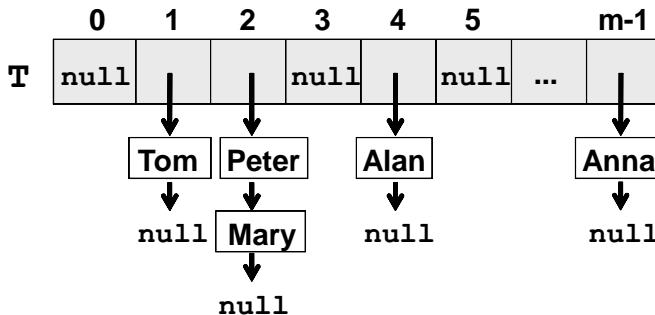
```
public override int GetHashCode()
{
  return 42;
}
```

If in a hash-table we use objects for keys from a class, that has the above implementation of **`GetHashCode()`**, it will have **very poor performance**, because every time, when we add a new element in the table, we would have to insert it **at the same place**. Every time we search the hash-table, we will encounter the same element.

In order to avoid the described behavior, we need the hash-function to **distribute the keys evenly** amongst the possible hash-code values.

Collisions with Hash-Functions

The situation where **two different keys have the same hash-code** is called **collision**. A good example of collision is shown below:

```
h("Alan")  =  4
h("Peter")= 2
h("Tom")  = 1
h("Mary")  = 2
h("Anna")  = 12
```

collision

We will look in more details how to solve the problem with collisions in the next paragraph. The simplest solution is obvious: order the pairs that have keys with the same hash-codes in a **list** or other data structure. Thus we don't solve the collisions but we accept them and we just put several key-value-pairs in the same element in the underlying array in the hash-table. This approach for collision resolution is known as **chaining**:

Therefore when using a constant 42 for hash-code our hash-table turns into a linear list and it becomes **very inefficient**.

Implementing the Method GetHashCode()

We will give a standard algorithm for implementing **GetHashCode()**, when this is necessarily:

First we need to choose which fields of the class will take part in the implementation of the **Equals(object)** method. This is necessary, because every time when **Equals()** returns **true**, the result from **GetHashCode()** should always return the same value.

This way the fields that do not take part in **Equals()**, should not take part in **GetHashCode()** as well.

After we choose which fields will take part for the calculation of **GetHashCode()**, we need to receive values from them (of type **int**). Here is a sample scheme:

- If the field is **bool**, for **true** we take **1**, and for **false** we take **0** (or directly call method **GetHashCode()** on **bool**).
- If the field is of type **int**, **byte**, **short**, **char**, we can convert it to **int**, with the cast operator (**int**) (or we could directly call **GetHashCode()**).
- If the field is type **long**, **float** or **double**, we could use the result from their own implementations of **GetHashCode()**.
- If the field is not a primitive type, we could call the method **GetHashCode()** of this object. If the field value is **null**, we can return **0**.
- If the field is an array or a collection, we take the hash-code from every element of this collection.

In the end we sum all the received **int** values, and before each addition we multiply the temporary result with a prime number (for example 83), while ignoring the eventual overflow of type **int**. For example, if we have 3 fields and their hash codes are **f1**, **f2** and **f3**, our hash function could combine them though the formula **hash** = (((**f1 * 83**) **+ f2**) *** 83)** + **f3**.

At the end we obtain a hash-code, which is very well distributed in the range of all 32-bit values. We can expect, that with a hash-code calculated this way, the **collisions would be rare**, because every change in some of the fields taking part in **GetHashCode()** leads to a major change in the hash code and thus reduces the chance for collision.

Implementing GetHashCode() – Example

Let's illustrate the above algorithm with an example. We have a class whose objects are presented as points in the three-dimensional space. The point will be represented with its coordinates in the three dimensional space **x**, **y** and **z**:

Point3D.cs

```
public class Point3D
{
  public double X { get; set; }
  public double Y { get; set; }
  public double Z { get; set; }

  public Point3D(double x, double y, double z)
  {
    this.X = x;
    this.Y = y;
    this.Z = z;
  }

  public override string ToString()
  {
    return String.Format("({0}, {1}, {2})",
      this.X, this.Y, this.Z);
  }
}
```

We can implement **GetHashCode()** easily using the above described algorithm that combines the hash values of the separate object fields:

```
public override bool Equals(object obj)
{
  if (this == obj)
    return true;
```

```
    Point3D other = obj as Point3D;

    if (other == null)
        return false;

    if (!this.X.Equals(other.X))
        return false;

    if (!this.Y.Equals(other.Y))
        return false;

    if (!this.Z.Equals(other.Z))
        return false;

    return true;
}

public override int GetHashCode()
{
    int prime = 83;
    int result = 1;
    unchecked
    {
        result = result * prime + X.GetHashCode();
        result = result * prime + Y.GetHashCode();
        result = result * prime + Z.GetHashCode();
    }

    return result;
}
```

This implementation is incomparably better, than returning a constant or just one of the fields or their sum. Although the **collisions might still happen, they would occur very rarely**.

Interface IEqualityComparer<T>

One of the most important things that we have learned so far is that in order to use instances of a class as keys for a dictionary, the class needs to properly implement **GetHashCode()** and **Equals(…)**. But what should we do if we want to use a class, that we cannot inherit or change? In this case the interface **IEqualityComparer<T>** comes to our aid.

It defines the following two operations:

- **bool Equals(T obj1, T obj2)** – returns **true** if **obj1** and **obj2** are equal
- **int GetHashCode(T obj)** – returns the hash-code of given object

As you might have already guessed, the dictionaries in .NET can use an instance of **IEqualityComparer<T>**, instead of using the corresponding methods of the given class that should be assigned for a key. This way the developers could use practically any class for a key of the dictionary, if they could assure **IEqualityComparer<T>** is implemented. Even more – when we pass **IEqualityComparer<T>** to a dictionary, we could change the way **GetHashCode()** and **Equals(…)** are calculated for every type, even for those built-in .NET Framework. This is because the dictionary uses interface methods instead of the corresponding methods of the class that is used for key. Here is an example of an implementation of **IEqualityComparer** for the class **Point3D** that we looked earlier:

```
public class Point3DEqualityComparer : IEqualityComparer<Point3D>
{
    public bool Equals(Point3D point1, Point3D point2)
    {
        if (point1 == point2)
            return true;

        if (point1 == null || point2 == null)
            return false;

        if (!point1.X.Equals(point2.X))
            return false;

        if (!point1.Y.Equals(point2.Y))
            return false;

        if (!point1.Z.Equals(point2.Z))
            return false;

        return true;
    }

    public int GetHashCode(Point3D obj)
    {
        Point3D point = obj as Point3D;
        if (point == null)
        {
            return 0;
        }

        int prime = 83;
```

```
        int result = 1;
        unchecked
        {
            result = result * prime + point.X.GetHashCode();
            result = result * prime + point.Y.GetHashCode();
            result = result * prime + point.Z.GetHashCode();
        }
        return result;
    }
}
```

Note that we implement both **Equals(…)** and **GetHashCode()**, not just **GetHashCode()** method.

Remember that the keys in hash-tables need to have correctly defined Equals(…) and GetHashCode() to work properly. This is not required for the values, just for the keys.

Always define both Equals(…) and GetHashCode(), never only one of them!

In order to use **Point3DEqualityComparer**, it's enough to pass it as argument to our dictionary's constructor. Here is an example:

```
static void Main()
{
    IEqualityComparer<Point3D> comparer =
        new Point3DEqualityComparer();
    Dictionary<Point3D, int> dict =
        new Dictionary<Point3D, int>(comparer);

    dict[new Point3D(4, 2, 5)] = 5;
    dict[new Point3D(1, 2, 3)] = 1;
    dict[new Point3D(3, 1, -1)] = 3;
    dict[new Point3D(1, 2, 3)] = 10;
    foreach (var entry in dict)
    {
        Console.WriteLine("{0} --> {1}", entry.Key, entry.Value);
    }
}
```

The result from the above code is:

```
(4, 2, 5) --> 5
(1, 2, 3) --> 10
(3, 1, -1) --> 3
```

We have 3 unique keys in the dictionary and the key (1, 2, 3) is used twice.

Resolving the Collision Problem

In practice, **collisions happen almost always**, excluding some rare and specific cases. That is why we need to live with the idea of the collisions presence in our hash-tables and take them into account. Let's have a look at several strategies for dealing with collisions.

Chaining in a List

The most widespread **method to resolve collisions** problem is called **chaining**. Its major concept consists of storing in a list all the pairs (key, value), which have the same hash-code for the key.

Implementation of a Dictionary with Hash-Table and Chaining

Let's have the task to **implement a dictionary data structure with a hash-table** and to **resolve the collisions by chaining**. With the example below we will show how it could be done. First, we are going to define a class, describing the **pair {key, value}**:

KeyValuePair.cs

```
/// <summary>A structure holding a pair {key, value}</summary>
/// <typeparam name="TKey">the type of the keys</typeparam>
/// <typeparam name="TValue">the type of the values</typeparam>
public struct KeyValuePair<TKey, TValue>
{
  /// <summary>Holds the key of the key-value pair</summary>
  public TKey Key { get; private set; }

  /// <summary>Holds the value of the key-value pair</summary>
  public TValue Value { get; private set; }

  /// <summary>Constructs a pair by given key + value</summary>
  public KeyValuePair(TKey key, TValue value) : this()
  {
    this.Key = key;
    this.Value = value;
  }

  /// <summary>Converts the key-value pair to a printable text.
  /// </summary>
  public override string ToString()
  {
```

```
    StringBuilder builder = new StringBuilder();
    builder.Append('[');
    if (this.Key != null)
    {
      builder.Append(this.Key.ToString());
    }
    builder.Append(", ");
    if (this.Value != null)
    {
      builder.Append(this.Value.ToString());
    }
    builder.Append(']');
    return builder.ToString();
  }
}
```

The class constructor has two parameters: key of type **TKey** and value of type **TValue**. There are defined two properties: one to access the key (**Key**) and another to access the value (**Value**). Note that these properties can only access the related members. There is no public functionality to change the key or value. This makes the class non-changeable (**immutable**). It is a good idea, because the objects, which will be kept inside the dictionary implementation, will be the same as these we will return as a result of a method for taking all the ordered pairs in the dictionary, for instance.

We have redefined the **ToString()** method in order to be able to easily print key-value pairs on the standard console output or in a text file.

Following is an example of a **generic dictionary interface**, which defines the most common operations of the data structure "dictionary":

IDictionary.cs

```
/// <summary>Interface that defines basic methods needed for a
/// "dictionary" class which maps keys to values</summary>
/// <typeparam name="K">Key type</typeparam>
/// <typeparam name="V">Value type</typeparam>
public interface IDictionary<K, V> :
  IEnumerable<KeyValuePair<K, V>>
{
  ///<summary>Finds the value mapped to the given key</summary>
  /// <param name="key">the key to be searched</param>
  /// <returns>value for the specified key if it presents,
  /// or null if there is no value with such key</returns>
  V Get(K key);
```

```
    /// <summary>Assigns the specified value to the specified key
    /// in the dictionary. If the key already exists, its value is
    /// replaced with the new value and the old value is returned
    /// </summary>
    /// <param name="key">Key for the new value</param>
    /// <param name="value">Value to be mapped to that key</param>
    /// <returns>the old (replaced) value for the specified key
    /// or null if the key does not exist</returns>
    V Set(K key, V value);

    /// <summary>Gets or sets the value of the entry in the
    /// dictionary identified by the key specified</summary>
    /// <remarks>A new entry will be created if the value is set
    /// for a key not currently in the dictionary</remarks>
    /// <param name="key">the key to identify the entry</param>
    /// <returns>the value of the entry in the dictionary
    /// identified by the provided key</returns>
    V this[K key] { get; set; }

    /// <summary>Removes an element in the dictionary identified
    /// by a specified key</summary>
    /// <param name="key">the key identifying the element to be
    /// removed</param>
    /// <returns>whether the element was removed or not</returns>
    bool Remove(K key);

    /// <summary>Returns the number of entries in the dictionary
    /// </summary>
    int Count { get; }

    /// <summary>Removes all the elements from the dictionary
    /// </summary>
    void Clear();
}
```

In the above defined **interface** as well as in the previous class, we use generics (template types), by which we define the parameters for the keys (**K**) and values (**V**). Such implementation allows us to use various data types for keys and values inside our dictionary. As we already know, the only requirement is to have proper definitions for **Equals()** and **GetHashCode()** methods inside the data type used for the keys.

Our interface **IDictionary<K, V>** looks much like the .NET standard interface **System.Collections.Generic.IDictionary<K, V>**, but it is simplified and describes only the most important operations of the "dictionary" data structure. It inherits the system interface **IEnumerable<DictionaryEntry<K,**

V>>, as doing so, the dictionary can be easily traversed by a simple **foreach** loop.

Following is an example of a **dictionary implementation** that uses **chaining** to handle collisions.

HashDictionary.cs

```
/// <summary>
/// Implementation of <see cref="IDictionary"/> interface
/// using a hash table. Collisions are resolved by chaining.
/// </summary>
/// <typeparam name="K">Type of the keys. Keys are required
/// to correctly implement Equals() and GetHashCode()
/// </typeparam>
/// <typeparam name="V">Type of the values</typeparam>
public class HashDictionary<K, V> : IDictionary<K, V>,
  IEnumerable<KeyValuePair<K, V>>
{
  private const int DEFAULT_CAPACITY = 16;
  private const float DEFAULT_LOAD_FACTOR = 0.75f;
  private List<KeyValuePair<K, V>>[] table;
  private float loadFactor;
  private int threshold;
  private int size;
  private int initialCapacity;

  /// <summary>Creates an empty hash table with the
  /// default capacity and load factor</summary>
  public HashDictionary()
    : this(DEFAULT_CAPACITY, DEFAULT_LOAD_FACTOR)
  { }

  /// <summary>Creates an empty hash table with given
  /// capacity and load factor</summary>
  public HashDictionary(int capacity, float loadFactor)
  {
    this.initialCapacity = capacity;
    this.table = new List<KeyValuePair<K, V>>[capacity];
    this.loadFactor = loadFactor;
    this.threshold = (int)(capacity * this.loadFactor);
  }

  /// <summary>Finds the chain of elements corresponding
  /// internally to given key (by its hash code)</summary>
  /// <param name="createIfMissing">creates an empty list
```

```
/// of elements if the chain still does not exist</param>
/// <returns>a list of elements in the chain or null</returns>
private List<KeyValuePair<K, V>> FindChain(
   K key, bool createIfMissing)
{
   int index = key.GetHashCode();
   index = index & 0x7FFFFFFF; // clear the negative bit
   index = index % this.table.Length;
   if (this.table[index] == null && createIfMissing)
   {
      this.table[index] = new List<KeyValuePair<K, V>>();
   }
   return this.table[index] as List<KeyValuePair<K, V>>;
}

/// <summary>Finds the value assigned to given key
/// (works extremely fast)</summary>
/// <returns>the value found or null when not found</returns>
public V Get(K key)
{
   List<KeyValuePair<K, V>> chain = this.FindChain(key, false);
   if (chain != null)
   {
      foreach (KeyValuePair<K, V> entry in chain)
      {
         if (entry.Key.Equals(key))
         {
            return entry.Value;
         }
      }
   }

   return default(V);
}

/// <summary>Assigns a value to certain key. If the key
/// exists, its value is replaced. If the key does not
/// exist, it is first created. Works very fast</summary>
/// <returns>the old (replaced) value or null</returns>
public V Set(K key, V value)
{
   if (this.size >= this.threshold)
   {
      this.Expand();
```

```
    }

    List<KeyValuePair<K, V>> chain = this.FindChain(key, true);
    for (int i = 0; i < chain.Count; i++)
    {
      KeyValuePair<K, V> entry = chain[i];
      if (entry.Key.Equals(key))
      {
        // Key found -> replace its value with the new value
        KeyValuePair<K, V> newEntry =
          new KeyValuePair<K, V>(key, value);
        chain[i] = newEntry;
        return entry.Value;
      }
    }
    chain.Add(new KeyValuePair<K, V>(key, value));
    this.size++;

    return default(V);
  }

  /// <summary>Gets / sets the value by given key. Get returns
  /// null when the key is not found. Set replaces the existing
  /// value or creates a new key-value pair if the key does not
  /// exist. Works very fast</summary>
  public V this[K key]
  {
    get
    {
      return this.Get(key);
    }
    set
    {
      this.Set(key, value);
    }
  }

  /// <summary>Removes a key-value pair specified
  /// by certain key from the hash table.</summary>
  /// <returns>true if the pair was found removed
  /// or false if the key was not found</returns>
  public bool Remove(K key)
  {
    List<KeyValuePair<K, V>> chain = this.FindChain(key, false);
```

```
    if (chain != null)
    {
        for (int i = 0; i < chain.Count; i++)
        {
            KeyValuePair<K, V> entry = chain[i];
            if (entry.Key.Equals(key))
            {
                // Key found -> remove it
                chain.RemoveAt(i);
                this.size--;
                return true;
            }
        }
    }
    return false;
}

/// <summary>Returns the number of key-value pairs
/// in the hash table (its size)</summary>
public int Count
{
    get
    {
        return this.size;
    }
}

/// <summary>Clears all ements of the hash table</summary>
public void Clear()
{
    this.table =
        new List<KeyValuePair<K, V>>[this.initialCapacity];
    this.size = 0;
}

/// <summary>Expands the underlying hash-table. Creates 2
/// times bigger table and transfers the old elements
/// into it. This is a slow (linear) operation</summary>
private void Expand()
{
    int newCapacity = 2 * this.table.Length;
    List<KeyValuePair<K, V>>[] oldTable = this.table;
    this.table = new List<KeyValuePair<K, V>>[newCapacity];
```

```
        this.threshold = (int)(newCapacity * this.loadFactor);
        foreach (List<KeyValuePair<K, V>> oldChain in oldTable)
        {
            if (oldChain != null)
            {
                foreach (KeyValuePair<K, V> keyValuePair in oldChain)
                {
                    List<KeyValuePair<K, V>> chain =
                        FindChain(keyValuePair.Key, true);
                    chain.Add(keyValuePair);
                }
            }
        }
    }

    /// <summary>Implements the IEnumerable<KeyValuePair<K, V>>
    /// to allow iterating over the key-value pairs in the hash
    /// table in foreach-loops</summary>
    IEnumerator<KeyValuePair<K, V>>
        IEnumerable<KeyValuePair<K, V>>.GetEnumerator()
    {
        foreach (List<KeyValuePair<K, V>> chain in this.table)
        {
            if (chain != null)
            {
                foreach (KeyValuePair<K, V> entry in chain)
                {
                    yield return entry;
                }
            }
        }
    }

    /// <summary>Implements IEnumerable (non-generic)
    /// as part of IEnumerable<KeyValuePair<K, V>></summary>
    IEnumerator IEnumerable.GetEnumerator()
    {
        return ((IEnumerable<KeyValuePair<K, V>>)this).
            GetEnumerator();
    }
}
```

We will pay attention to the most important points in this code. Let's begin with the constructor. The **public parameterless constructor** inside itself it

invokes another constructor, by passing some predefined values for **capacity** and **load factor**, which are used when the hash table is created.

Next thing, we pay attention to, is the actual **implementation of the hash table with chaining**. At the instantiation of the hash-table, inside the constructor we initialize an array of lists, which will contain any of our objects of type **KeyValuePair<K,V>**. We have created a private method **FindChain()**, for internal usage only, which **calculates the hash-code** of the key by calling **GetHashCode()** method and taking the modulus of the returned **hash-value** to the length of the table (**capacity**). Additionally the most-left bit is cleared to ensure the index is always a positive number. In that way the **index** of the current key in the internal table is calculated. The **list** of all the elements with the same hash-code is hold inside the internal table for this index. If the list is empty, it may have **null** as a value. Otherwise, at the specific index position there is a list of the elements for the specified key.

A special parameter is passed to the **FindChain()** method. This parameter indicates whether to create an empty list, if for the specific key there is no list of elements. It gives a kind of convenience for the methods of adding elements and resizing the hash-table.

The next thing, we pay attention to, is the **Expand()** method, which resizes the current internal table when the maximal allowed filling is reached. For this purpose we create a new table (array), with size twice as the current. Then we calculate a new value for the maximal allowed filling (the field **threshold**). Next coming is the most important part. We have extended the table and in this way we changed the value of **this.table.Length**. If we search for an element, which we have added already, the **FindChain(K key)** method will not return the correct chain at all, in which to search for it. That is why, we need to **transfer** all the elements of the old table, by not just copying the chains, but adding again all the **KeyValuePair<K,V>** objects into the newly created internal table of chains.

In order to implement the ability for iteration over the hash-table elements in **foreach**-loops, we have implemented the **IEnumerable<KeyValuePair<K, V>>** interface, which has **GetEnumerator()** method, returning an iterator (**IEnumerator**) of the elements of the hash-table. We simply iterate over the elements in the internal table and return them one at a time using the **yield return** C# keyword (it's is a complex concept explained in details in MSDN).

Now let's give an **example of how we can use our implementation of hash-table and its iterator**. We want to test whether the hash table copes correctly with collisions and with expanding, so we intentionally change the initial capacity of **3** and load factor of **0.9** when creating the hash table to ensure it **will resize soon** after few elements are put inside it. We first put an element, then read it, then overwrite its value, then read it again, then add a new element that causes a collision, then read it, then read the first element, then add an element causing the hash table to expand its internal array, etc. The code is given below and it is highly recommended to trace it

through the Visual Studio debugger and check at each step how the internal state of the hash table changes:

```
class PlayWithHashDictionary
{
  static void Main()
  {
    HashDictionary<Point3D, int> dict =
      new HashDictionary<Point3D, int>(3, 0.9f);

    dict[new Point3D(1, 2, 3)] = 1; // Put a key-value pair
    Console.WriteLine(dict[new Point3D(1, 2, 3)]); // Get value

    // Overwrite the previous value for the same key
    dict[new Point3D(1, 2, 3)] += 1;
    Console.WriteLine(dict[new Point3D(1, 2, 3)]);

    // Now this point will cause a collision with the
    // previous one and the elements will be chained
    dict[new Point3D(3, 2, 2)] = 42;

    Console.WriteLine(dict[new Point3D(3, 2, 2)]);

    // Test if the chaining works as expected, i.e.
    // elements with equal hash-codes are not overwritten
    Console.WriteLine(dict[new Point3D(1, 2, 3)]);

    // Creation of another entry in the internal table
    // This will cause the internal table to expand
    dict[new Point3D(4, 5, 6)] = 1111;
    Console.WriteLine(dict[new Point3D(4, 5, 6)]);

    // Delete an existing by its key
    dict.Remove(new Point3D(3, 2, 2));

    // Iterate through the dictionary entries and print them
    foreach (KeyValuePair<Point3D, int> entry in dict)
    {
      Console.WriteLine(
        "Key: " + entry.Key + "; Value: " + entry.Value);
    }
  }
}
```

As we could expect, the **result of the program execution** is the following:

```
1
2
42
2
1111
Key: (1, 2, 3); Value: 2
Key: (4, 5, 6); Value: 1111
```

Open Addressing Methods for Collision Resolution

Now let's look over the **methods for collision resolution**, alternative to chaining in a list. In general, the idea is, in case of collision we try to put the new pair in a table position, which is free. These methods differentiate from each other in the way they choose where to look for a free position for the new pair. Moreover, the new pair must be easily located at its new place.

Main drawback of this group of methods, compared to chaining in a list, is that they are inefficient at high rates of the load factor (close to 1).

Linear Probing

This is one of easiest methods for implementation. **Linear probing**, in general, can be presented with the following sample code:

```
int newPosition = (oldPosition + i) % capacity;
```

Here **capacity** is the internal table capacity, **oldPostion** is the position where collision occurs and **i** is a number for the next probing. If the new position is free, then we place the new pair there. Otherwise we try again (probing), incrementing **i**. Probing can be either forward or backwards. Backward probing is when instead of adding, we are subtracting **i** from the position we have collision for.

The advantage of this method is the **relatively quick way to find of a new position**. Unfortunately, if there was a collision at a certain place, there is an extremely high probability collision to occur again at the same place. So this, in practice, leads to a high inefficiency.

Using linear probing as a method for collision resolution in hash tables is inefficient and has to be avoided.

Quadratic Probing

Quadratic probing is a classic method for collision resolution. The main difference between **quadratic probing** and **linear probing** is that it uses a quadratic function of **i** (the number of the next probing) to find new position. Possible quadratic probing function is shown below:

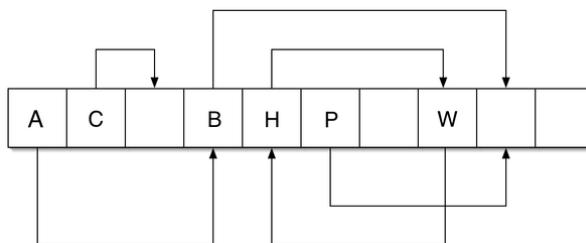
```
int newPosition = (oldPosition + c1*i + c2*i*i) % capacity;
```

The given example uses two constants: **c1** and **c2**, such that **c2** must not be 0, otherwise we are going back to linear probing.

By choosing **c1** and **c2** we define the position we are going to probe, compared to the starting position. For instance, if **c1** and **c2** are equal to 1, we are going to probe consequently **oldPosition**, **oldPosition + 2**, **oldPosition + 6**, … For a hash-table with capacity of the kind **2n**, the best is to choose **c1** and **c2** equal to **0.5**.

Quadratic probing is more efficient than linear the linear probing.

Double Hashing

As the name implies, the **double hashing** method uses **two different hash functions**. The main concept is that, the second hash function is used for the elements that fall into a collision. This method is better than the linear and quadratic probing, because all the next probing depends of the value of the key and not of the table position inside the hash-table. It makes sense, because the position of a given key depends on the current capacity of the hash-table.

Cuckoo Hashing

Cuckoo hashing is a relatively new method for collision resolution, using an open addressing. It was firstly presented by R. Pagh and F. Rodler in 2001. Its name comes from the behavior, observed with some kinds of cuckoos. The mother cuckoos push out the eggs and/or the nests out of other birds, in order to put their own eggs there and the other birds mistakenly care for the cuckoos' eggs in that way. (Also for the nests, after the incubation)

The main idea of this method is the use of **two hash-functions** instead of one. In this way, we have **not one, but two positions to place the element** inside the hash-table. If one of the positions is free, then we just put the element there. If both are taken, then we put the new element in one of them and it "**kicks out**" the element, which was already there. In turn, the "kicked" element is going to his alternative position and "kicks" another element out, if necessary. The new "kicked out" is repeating the procedure, and in that way until reaching a free position or we fall into a loop. In the last case, the whole hash table is built again with greater size and new hash-functions.

On the figure bellow it is shown an **example** scheme of a hash-table using cuckoo hashing. Every position, containing an element, has a link to the alternative position for the key inside. Now, let’s play out different situations of adding an element.

If, at least one of the two hash functions result is a free cell, there is no problem. We put the element in one of them. Let both hash functions result is a taken cell and we randomly have been choosing one of them.

Let’s assume that this is the cell, containing element A. The new element "kicks out" A from his place, A in turn goes to its alternative position and "kicks out" B from his place. The **alternative position** of B is free, so the adding is successfully completed.

Let’s assume, that the cell, the new element is trying to "**kick out**" an element, is the cell containing H. Then we have a loop, formed by H and W. In this case, a rebuild must be done using greater size, and new hash-functions.

In its simplest version this method has a constant access to its elements, even in the worst case, but this is valid with the constraint that the load factor is less than 0.5.

The use of three different hash-functions instead of two could result in an efficient upper limit of the load factor above 0.9.

Some researches show, the **cuckoo hashing and its modifications could be much more efficient than the widely spread today chaining in a list** and open addressing methods. Nevertheless, this method is still not well adopted in the industry and not used internally in .NET Framework. The main stopper is the need of **two hash func**tions, which means that the class **`System.Object`** should introduce two **`GetHashCode()`** methods.

The "Set" Data Structure

In this section we will look over the **abstract data structure "set"** and two of its typical implementations. We will explain their advantages and disadvantages and which of them should be preferred for different situations.

The Abstract Data Structure "Set"

Sets are **collections of unique elements** (without any repeating elements inside). In the .NET context, it means, for every set object, calling its **`Equals()`** method and passing another object from the set as an argument, will always result in **`false`**. Note that two different objects in .NET may be equal when compared by certain field and thus in the data structure "set" only one of them could be put.

Some sets allow their elements to be **`null`**, while others do not allow.

Besides not allowing the repetition of objects, another important thing, that distinguishes sets from lists and arrays, is that **the set element has no**

index. The elements of the set cannot be accessed by any key, as it is with dictionaries. **The elements themselves are the keys**.

The only way to access an object from a set is by having available either the object itself or another object, which is equal to it. That is why, in practice we access all the elements of a given set at once, while iterating, by using the **foreach** loop construct.

Set Implementations in .NET Framework

In .NET (version 4.0 and above) there is an **interface ISet<T> representing the ADT "set"** and it has two standard implementation classes:

- **HashSet<T>** – **hash-table** based implementation of set.
- **SortedSet<T> – red-black tree** based implementation of set.

Let's review both of them and see their strong and weak sides.

The main operations, defined by the **ISet<T>** interface (abstract data structure set), are the following:

- **bool Add(element) –** adding the **element** to the set and returning **false** if the element is already present inside the set, otherwise returning **true**.
- **bool Contains(element) –** checks if the set already contains the element passed as an argument. If yes, returns **true** as a result, otherwise returns **false**.
- **bool Remove(element)** – removes the **element** from the set. Returns Boolean if the element has been present inside the set.
- **void Clear()** – removes all the elements from the set.
- **void IntersectWith(Set other)** – inside the current set remain only the elements of the intersection of both sets – the result is a set, containing the elements, which are present in both sets at the same time – the set, calling the method and the **other**, passed as parameter.
- **void UnionWith(Set other)** – inside the current set remain only the elements of the sets union – the result is a set, containing the elements of either one or the other, or both sets.
- **bool IsSubsetOf(Set other)** – checks if the current set is a subset of the **other** set. Returns **true**, if yes and **false**, if no.
- **bool IsSupersetOf(Set other)** – checks if the **other** set is a subset of the current one. Returns **true**, if yes and **false**, if no.
- **int Count** – a property, which returns the current number of elements inside the set.

Implementation with Hash-Table – HashSet<T>

As we already mentioned, the hash-table implementation of set in .NET is the **HashSet<T>** class. This class, like **Dictionary<K, V>**, has constructors, by which we might pass a list of elements, as well as an **IEqualityComparer** implementation, mentioned earlier. They have the same semantics, because here we use a hash-table again.

Here is an example, which **demonstrates the use of sets** and the already described, operations: union and intersection:

```
using System;
using System.Collections.Generic;

class StudentListSetsExample
{
   static void Main()
   {
      HashSet<string> aspNetStudents = new HashSet<string>();
      aspNetStudents.Add("S. Jobs");
      aspNetStudents.Add("B. Gates");
      aspNetStudents.Add("M. Dell");

      HashSet<string> silverlightStudents =
         new HashSet<string>();
      silverlightStudents.Add("M. Zuckerberg");
      silverlightStudents.Add("M. Dell");

      HashSet<string> allStudents = new HashSet<string>();
      allStudents.UnionWith(aspNetStudents);
      allStudents.UnionWith(silverlightStudents);

      HashSet<string> intersectStudents =
         new HashSet<string>(aspNetStudents);
      intersectStudents.IntersectWith(silverlightStudents);

      Console.WriteLine("ASP.NET students: " +
         string.Join(", ", aspNetStudents));
      Console.WriteLine("Silverlight students: " +
         string.Join(", ", silverlightStudents));
      Console.WriteLine("All students: " +
         string.Join(", ", allStudents));
      Console.WriteLine(
         "Students in both ASP.NET and Silverlight: " +
         string.Join(", ", intersectStudents));
   }
```

```
}
```

And the **output** from the above code is:

```
ASP.NET students: S. Jobs, B. Gates, M. Dell
Silverlight students: M. Zuckerberg, M. Dell
All students: S. Jobs, B. Gates, M. Dell, M. Zuckerberg
Students in both ASP.NET and Silverlight: M. Dell
```

Pay attention that "**M. Dell**" is present in both sets, but inside the union it is present only once. That is, because, as we already explained, one element might be present at most once in a given set.

Implementation with Red-Black Tree – SortedSet<T>

The standard .NET class **SortedSet<T>** is a set, implemented by a **balanced search tree** (red-black tree). Because of this, its elements are internally kept in an **increasing order**. For that reason we can only add elements, which are **comparable**. We remind that in .NET it typically means the objects are instances of a class, implementing **IComparable<T>**. We would demonstrate the use of the **SortedSet<T>** class by the following example:

```
using System;
using System.Collections.Generic;

class SortedSetsExample
{
  static void Main()
  {
    SortedSet<string> bandsBradLikes =
      new SortedSet<string>(new[] {
        "Manowar", "Blind Guardian", "Dio", "Kiss",
        "Dream Theater", "Megadeth", "Judas Priest",
        "Kreator", "Iron Maiden", "Accept"
      });

    SortedSet<string> bandsAngelinaLikes =
      new SortedSet<string>(new[] {
        "Iron Maiden", "Dio", "Accept", "Manowar", "Slayer",
        "Megadeth", "Running Wild", "Grave Gigger", "Metallica"
      });

    Console.Write("Brad Pitt likes these bands: ");
    Console.WriteLine(string.Join(", ", bandsBradLikes));
```

```
      Console.Write("Angelina Jolie likes these bands: ");
      Console.WriteLine(string.Join(", ", bandsAngelinaLikes));

      SortedSet<string> intersectBands =
        new SortedSet<string>(bandsBradLikes);
      intersectBands.IntersectWith(bandsAngelinaLikes);

      Console.WriteLine(string.Format(
        "Does Brad Pitt like Angelina Jolie? {0}",
        intersectBands.Count >= 5 ? "Yes!" : "No!"));

      Console.Write(
        "Because Brad Pitt and Angelina Jolie both like: ");
      Console.WriteLine(string.Join(", ", intersectBands));

      SortedSet<string> unionBands =
        new SortedSet<string>(bandsBradLikes);
      unionBands.UnionWith(bandsAngelinaLikes);

      Console.Write(
        "All bands that Brad Pitt or Angelina Jolie like: ");
      Console.WriteLine(string.Join(", ", unionBands));
    }
}
```

And the **output** of the program execution is:

```
Brad Pitt likes these bands: Accept, Blind Guardian, Dio, Dream
Theater, Iron Maiden, Judas Priest, Kiss, Kreator, Manowar,
Megadeth
Angelina Jolie likes these bands: Accept, Dio, Grave Gigger,
Iron Maiden, Manowar, Megadeth, Metallica, Running Wild, Slayer
Does Brad Pitt like Angelina Jolie? Yes!
Because Brad Pitt and Angelina Jolie both like: Accept, Dio,
Iron Maiden, Manowar, Megadeth
All bands that Brad Pitt or Angelina Jolie like: Accept, Blind
Guardian, Dio, Dream Theater, Grave Gigger, Iron Maiden, Judas
Priest, Kiss, Kreator, Manowar, Megadeth, Metallica, Running
Wild, Slayer
```

As we may note, the elements in all set are always ordered, in comparison with **HashSet<T>**.

Exercises

1. Write a program that counts, in a given array of integers, **the number of occurrences of each integer**.

   Example: array = {3, 4, 4, 2, 3, 3, 4, 3, 2}

   2 → 2 times   3 → 4 times   4 → 3 times

2. Write a program to remove from a sequence all the integers, which appear **odd number of times**. For instance, for the sequence **{4, 2, 2, 5, 2, 3, 2, 3, 1, 5, 2, 6, 6, 6}** the output would be **{5, 3, 3, 5}**.

3. Write a program that counts **how many times each word** from a given text file **words.txt** appears in it. The result words should be **ordered by their number of occurrences** in the text.

   Example: "This is the TEXT. Text, text, text – THIS TEXT! Is this the text?"

   Result: is → 2, the → 2, this → 3, text → 6.

4. Implement a **DictHashSet<T>** class, based on **HashDictionary<K, V>** class, we discussed in the text above.

5. Implement a **hash-table**, maintaining triples (**key1**, **key2**, **value**) and allowing quick **search by the pair of keys** and adding of triples.

6. Implement a **hash-table**, allowing the maintenance of **more than one value** for a specific key.

7. Implement a **hash-table**, using "**cuckoo hashing**" with 3 hash-functions.

8. Implement the data structure **hash-table** in a class **HashTable<K,T>**. Keep the data in an array of **key-value pairs** (**KeyValuePair<K,T>[]**) with initial capacity of 16. Resole the collisions with **quadratic probing**. When the hash table load runs over 75%, perform resizing to 2 times larger capacity. Implement the following methods and properties: **Add(key, value)**, **Find(key) → value**, **Remove(key)**, **Count**, **Clear()**, **this[]** and **Keys**. Try to make the hash-table to support iterating over its elements with **foreach**.

9. Implement the data structure "**Set**" in a class **HashedSet<T>**, using your class **HashTable<K, T>** to hold the elements. Implement all standard set operations like **Add(T)**, **Find(T)**, **Remove(T)**, **Count**, **Clear()**, **union** and **intersect**.

10. We are given three sequences of numbers, defined by the formulas:

    - **f1(0) = 1; f1(k) = 2*f1(k-1) + 3; f1 = {1, 5, 13, 29, …}**
    - **f2(0) = 2; f2(k) = 3*f2(k-1) + 1; f2 = {2, 7, 22, 67, …}**
    - **f3(0) = 2; f3(k) = 2*f3(k-1) - 1; f3 = {2, 3, 5, 9, …}**

Write a program to find the **intersection and union of sets of sequences' elements** within the range [0; 100000]: f_1 * f_2; f_1 * f_3; f_2 * f_3; f_1 * f_2 * f_3; f_1 + f_2; f_1 + f_3; f_2 + f_3; f_1 + f_2 + f_3. Here + and * mean respectively union and intersection of sets.

11. * Define **TreeMultiSet<T>** class, which allows to keep a **set of elements, in increasing order** and to have **duplicates** of the elements. Implement operations adding of element, finding the number of occurrences, deletion, iterator, min / max element finding, min / max deletion. Implement the possibility to pass an external **Comparer<T>** for elements comparison.

12. * We are given a list of arriving and departing **schedule at a bus station**. Write a program, using the **HashSet<T>** class, which by given **interval (start, end)** returns the number of buses, which have arrived and departed during that time. Example:

    We have the data of the following buses: [08:24-08:33], [08:20-09:00], [08:32-08:37], [09:00-09:15]. We are given the range [08:22-09:05]. The number of buses, arriving and departing during that time is 2.

13. * We are given a sequence **P** containing **L** integers **L** (1 < **L** < 50,000) and a number **N**. We call a "**lucky sub-sequence within P**" every sub-sequence of integers from **P** with a sum equal to **N**.

    Imagine we have a sequence **S**, holding all the lucky sub-sequences of **P**, kept in **decreasing order by their length**. When the length is the same, the sequences are ordered in **decreasing order by their elements**: from the leftmost to the rightmost. Write a program to return the **first 10 elements of S**.

    Example: We are given **N** = 5 and the sequence **P** = **{1, 1, 2, 1, -1, 2, 3, -1, 1, 2, 3, 5, 1, -1, 2, 3}**. The sequence **S** consists of the following 13 sub-sequences of **P**:

    - **[1, -1, 2, 3, -1, 1]**
    - **[1, 2, 1, -1, 2]**
    - **[3, -1, 1, 2]**
    - **[2, 3, -1, 1]**
    - **[1, 1, 2, 1]**
    - **[1, -1, 2, 3]**
    - **[1, -1, 2, 3]**
    - **[-1, 1, 2, 3]**
    - **[5, 1, -1]**
    - **[2, 3]**
    - [2, 3]
    - [2, 3]
    - [5]

The last 10 elements of **P** are given in bold.

Solutions and Guidelines

1. Use **Dictionary<TKey, TValue> counts** and though a single scan through the input numbers count the occurrences of each one. When you pass through a number **p**, if it is missing in the dictionary **counts[p] = 1**. If the number is already stored in the dictionary, increase its count: **counts[p] = counts[p] + 1**. Finally scan through the element of the dictionary (with **foreach**-loop) and **print its key-value pairs**.

2. Use **Dictionary<K, T>** to count how many times each element occurs (like in the previous problem) and **List<T>** where you can add all elements occurring even number of times.

3. Use **Dictionary<string, int>** with word as a key and number of occurrences as a value. After **counting all the words**, **sort the dictionary by value** using something like this:

```
var sorted = dictionary.OrderBy(p => p.Value);
```

   To use the **OrderBy(<keySelector>)** extension method you need to include the **System.Linq** namespace.

4. Use the element of the set as **key and value at the same time**.

5. Use **hash-table of hash-tables**: **Dictionary<key, Dictionary<key, value>>**. Think about how to add and search elements in this structure.

6. Use **Dictionary<K, List<V>>**.

7. You can use **GetHashCode() % size** as the first hash-function, **(GetHashCode() * 83 + 7) % size** as the second, (**GetHashCode () * GetHashCode() + 19) % size**) as the third.

8. **Follow the example from the section "[Implementation of a Dictionary with Hash-Table and Chaining]"**. Read about quadratic probing in Wikipedia: http://en.wikipedia.org/wiki/Quadratic_probing. In order to expand the hash table (double its size), you can allocate an array with double size, transfer all the elements from the old one to the new one and at the end redirect the reference from the old array to the new one. To have **foreach** on your collection, implement the interface **IEnumerable** and inside your **GetEnumerator()** method you must return **GetEnumerator()** to the array of lists. You can use **yield** operator.

9. One way to solve the task is to use as key for the hash-table the element of the set and **as a value always true**. The union and intersection will be done by looping all the elements of the first set and checking if there is (respectively there is not) element of the second one.

10. Find all the members of the three sequences inside the given range and using **HashSet<int>** implement union and intersection of sets after that. At the end do the calculations requested.

11. **TreeMultiSet<T>** class you can implement by using the .NET system class **SortedDictionary<K, List<T>>**. It is not so easy, so take enough time to write the code and test it.

12. The obvious solution is to check for all the buses whether they arrive or depart in the given range. But according to the task terms we have to use **HashSet<T>**. Let's think how.

    With a linear scan (a **for**-loop) we can find **all buses arriving after the beginning** of the range and find **all buses departing before the end** of the range. These are **two separate sets**, right? The intersection of these sets should give us the set of buses we need.

    If **TimeInterval** is a class, keeping the schedule of a bus (**arriveHour**, **arriveMinute**, **departureHour**, **departureMinute**), the intersection could be efficiently found by **HashSet<TimeInterval>** with correctly defined **GetHashCode()** and **Equals()**.

    Another, **efficient solution** is to use **SortedSet<T>** and its method **GetViewBetween(<start>, <end>)**, but this contradicts to the problem description (recall that we are assigned to use **HashSet<T>**).

13. The first idea for a solution is simple: Using **two nested loops we find all lucky sub-sequences** of the sequence **P**. After that we **sort them** by their length (and by their elements as second criteria) and at the end we print the first 10. However, this algorithm will not work well if we have millions of sub-sequences. It may cause "**out of memory**".

    We would describe an idea for a **much efficient solution**: we will first define a class **Sequence<T>** to hold a sequence of elements. We will implement **IComparable<Sequence<T>>** to compare sequences by length in decreasing order (and by elements in decreasing order when the length is the same).

    Later we will use our **TreeMultiSet<T>** class. Inside we will **keep the first 10 sub-sequences of S**, i.e. multi-set of the lucky sub-sequences of **P**, kept in decreasing order by length (and in decreasing order of their content when the length is the same). When we have 10 sub-sequences inside the multi-set and we add 11th sequence, it would **take its correct place** in the order, because of the **IComparable<Sequence<T>>** defined. After that we can delete the 11th subsequence, because it is not amongst the first 10. In that way we would **always keep the first 10 elements**, discarding the others in any given moment, consuming **much less memory** and with no need of sorting at the end. The implementation is not so easy, so spare enough time for it.

Chapter 19. Data Structures and Algorithm Complexity

In This Chapter

In this chapter we will **compare the data structures** we have learned so far by the **performance** (execution speed) of the basic operations (addition, search, deletion, etc.). We will give specific tips in what situations **what data structures to use**. We will explain how to choose between data structures like hash-tables, arrays, dynamic arrays and sets implemented by hash-tables or balanced trees. Almost all of these structures are implemented as part of NET Framework, so to be able to **write efficient and reliable code** we have to learn to apply the most appropriate structures for each situation.

Why Are Data Structures So Important?

You may wonder why we pay so **much attention to data structures** and why we review them in such a great details. The reason is we aim to make out of you **thinking** software engineers. Without knowing the basic data structures and computer algorithms in programming well, you cannot be good developers and risk to stay an amateur. Whoever knows **data structures and algorithms** well and starts thinking about their correct use has big chance to become a professional – one that analyzes the problems in depth and proposes efficient solutions.

There are hundreds of books written on this subject. In the four volumes, named **"The Art of Computer Programming", Donald Knuth** explains data structures and algorithms in more than 2500 pages. Another author, **Niklaus Wirth**, has named his book after the answer to the question "why are data structures so important", which is "**Algorithms + Data Structures = Programs**". The main theme of the book is again the fundamental algorithms and data structures in programming.

	Data structures and algorithms are the fundamentals of programming. In order to become a good developer it is essential to master the basic data structures and algorithms and learn to apply them in the right way.

To a large degree our book is focused on learning data structures and algorithms along with the programming concepts, language syntax and problem solving. We also try to illustrate them in the context of modern software engineering with C# and .NET Framework.

Algorithm Complexity

We cannot talk about **efficiency of algorithms and data structures** without explaining the term "algorithm complexity", which we have already mentioned several times in one form or another. We will avoid the mathematical definitions and we are going to give a simple explanation of what the term means.

Algorithm complexity is a **measure** which evaluates the order of the **count of operations**, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough **approximation of the number of steps** necessary to execute an algorithm. When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order.

Algorithm complexity is commonly represented with the **O(f) notation**, also known as **asymptotic notation** or "**Big O notation**", where **f** is the function of the size of the input data. The asymptotic computational complexity **O(f)** measures the order of the consumed resources (CPU time, memory, etc.) by certain algorithm expressed as function of the input data size.

Complexity can be **constant**, **logarithmic**, **linear**, **n*log(n)**, **quadratic**, **cubic**, **exponential**, etc. This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of "**algorithms complexity**" or just "**complexity**" we use the term "**running time**".

Algorithm complexity is a rough approximation of the number of steps, which will be executed depending on the size of the input data. Complexity gives the order of steps count, not their exact count.

Typical Algorithm Complexities

This table will explain what every type of complexity (running time) means:

Complexity	Running Time	Description
constant	O(1)	It takes a **constant number of steps** for performing a given operation (for example 1, 5, 10 or other number) and this count does not depend on the size of the input data.
logarithmic	O(log(N))	It takes the order of **log(N) steps**, where the base of the logarithm is most often 2, for performing a given operation on N elements. For example, if N = 1,000,000, an algorithm with a complexity O(log(N))

		would do about 20 steps (with a constant precision). Since the base of the logarithm is not of a vital importance for the order of the operation count, it is usually omitted.
linear	O(N)	It takes nearly the **same amount of steps as the number of elements** for performing an operation on N elements. For example, if we have 1,000 elements, it takes about 1,000 steps. Linear complexity means that the number of elements and the number of steps are linearly dependent, for example the number of steps for N elements can be N/2 or 3*N.
	O(n*log(n))	It takes **N*log(N) steps** for performing a given operation on N elements. For example, if you have 1,000 elements, it will take about 10,000 steps.
quadratic	$O(n^2)$	It takes the order of **N^2 number** of steps, where the N is the size of the input data, for performing a given operation. For example if N = 100, it takes about 10,000 steps. Actually we have a quadratic complexity when the number of steps is in quadratic relation with the size of the input data. For example for N elements the steps can be of the order of $3*N^2/2$.
cubic	$O(n^3)$	It takes the order of **N^3 steps**, where N is the size of the input data, for performing an operation on N elements. For example, if we have 100 elements, it takes about 1,000,000 steps.
exponential	$O(2^n)$, O(N!), $O(n^k)$, …	It takes a number of steps, which is with an **exponential** dependability with the size of the input data, to perform an operation on N elements. For example, if N = 10, the exponential function 2^N has a value of 1024, if N = 20, it has a value of 1 048 576, and if N = 100, it has a value of a number with about 30 digits. The exponential function N! grows even faster: for N = 5 it has a value of 120, for N = 10 it has a value of 3,628,800 and for N = 20 – 2,432,90,008,176,640,000.

When evaluating complexity, **constants are not taken into account**, because they do not significantly affect the count of operations. Therefore an algorithm which does N steps and algorithms which do N/2 or 3*N respectively are considered linear and approximately equally efficient, because they perform a number of operations which is of the same order.

Complexity and Execution Time

The **execution speed** of a program depends on the complexity of the algorithm, which is executed. If this complexity is low, the program will execute fast even for a big number of elements. If the complexity is high, the program will execute slowly or will not even work (it will hang) for a big number of elements.

If we take an average computer from 2008, we can assume that it can perform about **50,000,000 elementary operations per second**. This number is a rough approximation, of course. The different processors work with a different speed and the different elementary operations are performed with a different speed, and also the computer technology constantly evolves. Still, if we accept we use an average home computer from 2008, we can make the following conclusions about the **speed of execution** of a given program depending on the algorithm complexity and size of the input data.

Algorithm	10	20	50	100	1,000	10,000	100,000
O(1)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n*log(n))	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.
O(n^2)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	2 sec.	3-4 min.
O(n^3)	< 1 sec.	< 1 sec.	< 1 sec.	< 1 sec.	20 sec.	5.55 hours	231.5 days
O(2^n)	< 1 sec.	< 1 sec.	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 sec.	hangs	hangs	hangs	hangs	hangs	hangs
O(n^n)	3-4 min.	hangs	hangs	hangs	hangs	hangs	hangs

We can draw many **conclusions** from the above table:

- Algorithms with a **constant**, **logarithmic** or **linear complexity** are so **fast** that we cannot feel any delay, even with a relatively big size of the input data.
- Complexity **O(n*log(n))** is **similar to the linear** and works nearly as fast as linear, so it will be very difficult to feel any delay.
- **Quadratic** algorithms work very well up to several thousand elements.
- **Cubic** algorithms work well if the elements are not more than 1,000.
- Generally these so called **polynomial algorithms** (any, which are not exponential) are considered to be fast and working well for thousands of elements.
- Generally the **exponential algorithms do not work well** and we should avoid them (when possible). If we have an exponential solution to a task, maybe we actually do not have a solution, because it will work only if the number of the elements is below 10-20. Modern cryptography is based exactly on this – there are not any fast (non-exponential) algorithms for finding the secret keys used for data encryption.

	If you solve a given problem with an exponential complexity this means that you have solved it for a small amount of input data and generally your solution does not work.

The data in the table is just for orientation, of course. Sometimes a **linear algorithm could work slower than a quadratic** one or a cubic algorithm could work faster than O(n*log(n)). The reasons for this could be many:

- It is possible the **constants** in an algorithm with a low complexity to be big and this could eventually make the algorithm slow. For example, if we have an algorithm, which makes **50*n steps** and another one, which makes **1/100*n*n** steps, for elements up to 5000 the quadratic algorithm will be faster than the linear.
- Since the complexity evaluation is made in the **worst case scenario**, it is possible a quadratic algorithm to work better than O(n*log(n)) in 99% of the cases. We can give an example with the algorithm **`QuickSort`** (the standard sorting algorithm in .NET Framework), which in the **average case** works a bit better than **`MergeSort`**, but in the worst case **`QuickSort`** can make the order of **n^2 steps**, while **`MergeSort`** does always **O(n*log(n)) steps**.
- It is possible an algorithm, which is evaluated to execute with a linear complexity, to not work so fast, because of an **inaccurate complexity evaluation**. For example if we search for a given word in an array of words, the complexity is **linear**, but at every step **string comparison** is performed, which is not an elementary operation and can take much more time than performing simple elementary operation (for example comparison of two integers).

Complexity by Several Variables

Complexity can depend on several input variables at once. For example, if we look for an element in a rectangular **matrix with sizes M and N**, the searching speed depends on M and N. Since in the worst case we have to traverse the entire matrix, we will do M*N number of steps at most. Therefore the complexity is **O(M*N)**.

Best, Worst and Average Case

Complexity of algorithms is usually evaluated in the **worst case** (most unfavorable scenario). This means in the average case they can work faster, but in the worst case they work with the evaluated complexity and not slower.

Let's take an example: **searching in array**. To find the searched key in the **worst case**, we have to check all the elements in the array. In the **best case** we will have luck and we will find the element at first position. In the average case we can expect to check half the elements in the array until we find the one we are looking for. Hence in the **worst case** the complexity is O(N) – **linear**. In the average case the complexity is O(N/2) = O(N) – linear, because when evaluating complexity one does not take into account the constants. In the best case we have a constant complexity O(1), because we make only one step and directly find the element.

Roughly Estimated Complexity

Sometimes it is **hard to evaluate the exact complexity** of a given algorithm, because it performs operations and it is not known exactly how much time they will take and how many operations will be done internally. Let's take the example of **searching a given word in an array of strings** (texts). The task is easy: we have to traverse the array and search in every text with **`Substring()`** or with a regular expression for the given word. We can ask ourselves the question: if we had 10,000 texts, would this work fast? What if the texts were 100,000? If we carefully think about it, we will implement that in order to evaluate adequately, we have to **know how big are the texts**, because there is a difference between searching in people's names (which are up to 50-100 characters) and searching in scientific articles (which are roughly composed by 20,000 – 30,000 characters). However, we can evaluate the complexity using the length of the texts, through which we are searching: it is at least **O(L)**, where L is the sum of the lengths of all texts. This is a pretty rough evaluation, but it is much more accurate than complexity O(N), where N is the number of the texts, right? We should think whether we take into account all situations, which could occur. Does it matter **how long the searched word is**? Probably searching of long words is slower than searching of short words. In fact things are slightly different. If we search for "**aaaaaaa**" in the text "**aaaaaabaaaaacaaaaaabaaaaacaaaaab**", this will be slower than if we search for "xxx" in the same text, because in the first case we will get more sequential matches than in the second case. Therefore,

in some special situations, searching seriously depends on the length of the word we search and the complexity O(L) could be underestimated.

Complexity by Memory

Besides the number of steps using a function of the input data, one can **measure other resources**, which an algorithm uses, for example **memory**, count of disk operations, etc. For some algorithms the execution speed is not as important as the **memory they use**. For example if a given algorithm is linear but it uses RAM in the order of N^2, it will be probably shortage of memory if N = 100,000 (then it will need memory in order of 9 GB RAM), despite the fact that it should work very fast.

Estimating Complexity – Examples

We are going to give **several examples**, which show how you can estimate the complexity of your algorithms, and decide whether the code written by you will work fast:

If we have a single loop from 1 to N, its complexity is **linear – O(N)**:

```
int FindMaxElement(int[] array)
{
   int max = int.MinValue;
   for (int i = 1; i < array.Length; i++)
   {
      if (array[i] > max)
      {
         max = array[i];
      }
   }
   return max;
}
```

This code will work well even if the number of elements is huge.

If we have **two of nested loops from 1 to N**, their complexity is **quadratic – O(N^2)**. Example:

```
int FindInversions(int[] array)
{
   int inversions = 0;
   for (int i = 0; i < array.Length - 1; i++)
   {
      for (int j = i + 1; j < array.Length; j++)
      {
         if (array[i] > array[j])
         {
```

```
                inversions++;
            }
        }
    }
    return inversions;
}
```

If the elements are no more than several thousand or tens of thousands, this code will work well.

If we have **tree nested loops from 1 to N**, their complexity is **cubic – $O(N^3)$**. Example:

```
long Sum3(int n)
{
    long sum = 0;
    for (int a = 1; a < n; a++)
    {
        for (int b = 1; b < n; b++)
        {
            for (int c = 1; c < n; c++)
            {
                sum += a * b * c;
            }
        }
    }
    return sum;
}
```

This code will work well if the number of elements is below 1,000.

If we have **two nested loops from 1 to N and from 1 to M** respectively, their complexity will be **quadratic – O(N*M)**. Example:

```
long SumMN(int n, int m)
{
    long sum = 0;
    for (int x = 1; x <= n; x++)
    {
        for (int y = 1; y <= m; y++)
        {
            sum += x * y;
        }
    }
    return sum;
}
```

The speed of this code **depends on two variables**. The code will work well if M, N < 10,000 or if at least the one variable has a value small enough.

We should pay attention to the fact that **not always** tree nested loops mean cubic complexity. Here is an example in which the complexity is **O(N*M)**:

```
long SumMN(int n, int m)
{
   long sum = 0;
   for (int x = 1; x <= n; x++)
   {
      for (int y = 1; y <= m; y++)
      {
         if (x == y)
         {
            for (int i = 1; i <= n; i++)
            {
               sum += i * x * y;
            }
         }
      }
   }
   return sum;
}
```

In this example the most inner loop executes exactly min(M, N) times and does not significantly affect the algorithm speed. The outer code executes approximately N*M + min(M,N)*N steps, i.e. its complexity is **quadratic**.

When using a **recursion**, the **complexity is more difficult to be estimated**. Here is an example:

```
long Factorial(int n)
{
   if (n == 0)
   {
      return 1;
   }
   else
   {
      return n * Factorial(n - 1);
   }
}
```

In this example the complexity is obviously linear – O(N), because the function **`Factorial()`** executes exactly once for each of the numbers 1 … n.

Here is a recursive function for which it is very **difficult to estimate** the complexity:

```
long Fibonacci(int n)
{
   if (n == 0)
   {
      return 1;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return Fibonacci(n - 1) + Fibonacci(n - 2);
   }
}
```

If we write down what really happens when the upper code executes, we will see that the function calls itself as many times as the Fibonacci's n+1 number is. We can roughly evaluate the complexity by another way too: since on every step of the function execution 2 recursive calls are done in average, the count of the recursive calls must be **in order of 2^n**, i.e. we have an **exponential complexity**. This automatically means that for values greater than 20-30 the function will "hang". You may check this yourself.

The same function for calculating the **n^{th} number of Fibonacci** can be written with a **linear complexity** in the following way:

```
long Fibonacci(int n)
{
   long fn = 1;
   long fn1 = 1;
   long fn2 = 1;
   for (int i = 2; i < n; i++)
   {
      fn = fn1 + fn2;
      fn2 = fn1;
      fn1 = fn;
   }
   return fn;
}
```

You see that the complexity estimation helps us to **predict whether a given code will work slowly** before we have run it and it implies we should look for a more efficient solution.

Comparison between Basic Data Structures

After you have been introduced to the term algorithm complexity, we are now ready to make a **comparison between the basic data structures**, which we know from the last few chapters, and to estimate with what complexity each of them performs the basic operations like **addition**, **searching**, **deletion** and **access by index** (when applicable). In that way we could easily judge according to the operations we expect to need, **which structure would be the most appropriate**. The complexities of the basic operations on the basic data structures, which we have reviewed in the previous chapters, are given in the table below:

Data structure	Addition	Search	Deletion	Access by index
Array (**T[]**)	O(N)	O(N)	O(N)	O(1)
Linked list (**LinkedList<T>**)	O(1)	O(N)	O(N)	O(N)
Dynamic array (**List<T>**)	O(1)	O(N)	O(N)	O(1)
Stack (**Stack<T>**)	O(1)	-	O(1)	-
Queue (**Queue<T>**)	O(1)	-	O(1)	-
Dictionary, implemented with a hash-table (**Dictionary<K, T>**)	O(1)	O(1)	O(1)	-
Dictionary, implemented with a balanced search tree (**SortedDictionary<K, T>**)	O(log(N))	O(log(N))	O(log(N))	-
Set, implemented with a hash-table (**HashSet<T>**)	O(1)	O(1)	O(1)	-
set, implemented with a balanced search tree (**SortedSet<T>**)	O(log(N))	O(log(N))	O(log(N))	-

We let the reader to think about how these complexities were estimated.

When to Use a Particular Data Structure?

Let's skim through all the structures in the table above and explain **in what situations we should use them** as well as how their complexities are evaluated.

Array (T[])

The arrays are **collections of fixed number of elements** from a given type (for example numbers) where the **elements preserved their order**. Each

element can be accessed through its index. The arrays are memory areas, which have a **predefined size**.

Adding a new element in an array is a **slow operation**. To do this we have to allocate a memory with the same size plus one and copy all the data from the original array to the new one.

Searching in an array **takes time** because we have to compare every element to the searched value. It takes N/2 comparisons in the average case.

Removing an element from an array is a **slow operation**. We have to allocate a memory with the same size minus one and copy all the old elements except the removed one.

Accessing by index is direct, and thus, a **fast operation**.

The arrays should be used only when we have to p**rocess a fixed number of elements** to which we need a **quick access by index**. For example, if we have to sort some numbers, we can keep them in an array and then apply some of the well-known sorting algorithms. If we have to change the elements' count, the array is not the correct data structure we should use.

Use arrays when you have to process a fixed number of elements to which you need an access through index.

Singly / Doubly Linked List (LinkedList<T>)

Singly and doubly **linked lists** hold **collection of elements**, which preserve their order. Their representation in the memory is dynamic, pointer-based. They are linked sequences of element.

Adding is a fast operation but it is a bit slower than adding to a `List<T>` because every time when we add an element to a linked list we allocate a new memory area. The memory allocation works at speed, which cannot be easily predicted.

Searching in a linked list is a **slow operation** because we have to traverse through all of its elements.

Accessing an element by index is a slow operation because there is **no indexing** in singly and doubly linked lists. You have to go through all the elements from the start one by one instead.

Removing an element at a specified index is a **slow operation** because reaching the element through its index is a slow operation. Removing an element with a specified value is a slow operation too, because it involves searching.

Linked list can **quickly add and remove elements** (with a constant complexity) **at its two ends** (head and tail). Hence, it is very handy for an implementation of stacks, queues and similar data structures.

Linked lists are **rarely used** in practice because the dynamic arrays (**List<T>**) can do almost exact same operations **LinkedList** does, plus for the most of them it works faster and more comfortable.

When you need a linked list, use **List<T>** instead of **LinkedList<T>**, because it doesn't work slower and it gives you better speed and flexibility. Use **LinkedList** when you have to **add and remove elements at both ends** of the data structure.

When you need to add and remove elements at both ends of the list, use LinkedList<T>. Otherwise use List<T>.

Dynamic Array (List<T>)

Dynamic array (**List<T>**) is one of the **most popular data structures** used in programming. It does not have fixed size like arrays, and allows direct access through index, unlike linked lists (**LinkedList<T>**). The dynamic array is also known as "array list", "resizable array" and "dynamic array".

List<T> holds its elements in an array, which has a bigger size than the count of the stored elements. Usually when we **add an element**, there is an empty cell in the list's inner array. Therefore this operation takes a **constant time**. Occasionally the array has been filled and it has to expand. This takes linear time, but it rarely happens. If we have a large amount of additions, the average-case complexity of adding an element to **List<T>** will a constant – O(1). If we sum the steps needed for adding 100,000 elements (for both cases – "fast add" and "add with expand") and divide by 100,000, we will obtain a constant which will be nearly the same like for adding 1,000,000 elements.

This statistically-averaged complexity calculated for large enough amount of operations is called **amortized complexity**. Amortized linear complexity means that if we add 10,000 elements consecutively, the overall count of steps will be of the order of 10,000. In most cases add it will execute in a constant time, while very rarely adding will execute in linear time.

Searching in **List<T>** is a **slow operation** because you have to traverse through all the elements.

Removing by index or value executes in a linear time. It is a **slow operation** because we have to move all the elements after the deleted one with one position to the left.

The **indexed access** in **List<T>** is instant, in a **constant time**, since the elements are internally stored in an array.

Practically **List<T> combines the best of arrays and lists**, for which it is a preferred data structure in many situations. For example if we have to process a text file and to extract from it all words (with duplicates), which

match a regular expression, the most suitable data structure in which we can accumulate them is **List<T>**, because we need a list, the length of which is unknown in advance and can grow dynamically.

The dynamic array (**List<T>**) is appropriate, when we have to add elements frequently as well as keeping their order of addition and access them through index. If we often we have to search or delete elements, **List<T>** is not the right data structure.

	Use List<T>**, when you have to add elements quickly and access them through index.**

Stack

Stack is a linear data structure in which there are 3 operations defined: adding an element at the top of the stack (**push**), removing an element from the top of the stack (**pop**) and inspect the element from the top without removing it (**peek**). All these operations are **very fast** – it takes a **constant time** to execute them. The stack does not support the operations search and access through index.

The stack is a data structure, which has a **LIFO behavior** (last in, first out). It is used when we have to model such a behavior – for example, if we have to keep the path to the current position in a recursive search.

	Use a stack when you have to implement the behavior "last in, first out" (LIFO).

Queue

Queue is a linear data structure in which there are two operations defined: adding an element to the tail (**enqueue**) and extract the front-positioned element from the head (**dequeue**). These two operations take a **constant time** to execute, because the queue is usually implemented with a linked list. We remind that the linked list can quickly add and remove elements from its both ends.

The queue’s behavior is **FIFO (first in, first out)**. The operations searching and accessing through index are not supported. Queue can naturally model a list of waiting people, tasks or other objects, which have to be processed in the same order as they were added (enqueued).

As an example of **using a queue** we can point out the implementation of the **BFS (breadth-first search) algorithm**, in which we start from an initial element and all its neighbors are added to a queue. After that they are processed in the order they were added and their neighbors are added to the queue too. This operation is repeated until we reach the element we are looking for or we process all elements.

Use a queue when you have to implement the behavior "first in, first out" (FIFO).

Dictionary, Implemented with a Hash-Table (Dictionary<K, T>)

The data structure "dictionary" suggests **storing key-value pairs** and provides a **quick search by key**. The implementation with a hash table (the class **Dictionary<K,T>** in .NET Framework) has a **very fast add, search and remove** of elements – **constant complexity** at the average case. The operation access through index is not available, because the elements in the hash-table have no order, i.e. an almost **random order**.

Dictionary<K,T> keeps internally the elements in an **array** and puts every element at the position calculated by the hash-function. Thus the array is partially filled – in some cells there is a value, others are empty. If more than one element should be placed in a single cell, elements are stored in a linked list. It is called **chaining**. This is one of the few ways to resolve the collision problem. When the load factor exceeds 75%, the size is doubled and all the elements occupy new positions. This operation has a linear complexity, but it is executed so rarely, that the amortized complexity remains a constant.

Hash-table has one peculiarity: if we choose a **bad hash-function** causing many collisions, the basic operations can become **very inefficient** and reach linear complexity. In practice, however, this hardly happens. Hash-table is considered to be the fastest data structure, which provides adding and searching by key.

Hash-table in .NET Framework permits **each key to be put only once**. If we add two elements with the same key consecutively, the last will replace the first and we will eventually lose an element. This important feature should be considered.

From time to time one key will have to keep multiple values. This is not standardly supported but we can store the values matching this key in a **List<T>** as a sequence of elements. For example if we need a hash-table **Dictionary<int, string>**, in which to accumulate pairs {integer, string} with duplicates, we can use **Dictionary<int, List<string>>**. Some external libraries have ready to use data structure called **MultiDictionary<K,V>**.

Hash-table is **recommended** to be used every time we need **fast addition** and **fast search by key**. For example if we have to count how many times each word is encountered in a set of words in a text file, we can use **Dictionary<string, int>** – the key will be a particular word, the value – how many times we have seen it.

Use a hash-table, when you want to add and search by key very fast.

A lot of programmers (mostly beginners) live with the delusion the main advantage of using a hash-table is the comfort of **searching a value by its key**. Actually this is wrong. We can implement searching a key with an array, a list or even a stack. There is no problem, everyone can build it. We can define a class **Entry**, which holds a key-value pair and after that we will work with an array or a list with **Entry** elements. We can implement the search but by any circumstances it will work slowly. This is the big problem with lists and arrays – they do not offer a fast search. Unlike them the hash-table can **search and add new elements very fast**.

The main advantage of the hash-table over the other data structures is a very quick searching and addition. The comfort for the developers is a secondary factor.

Dictionary, Implemented with a Balanced Tree (SortedDictionary<K,T>)

The implementation of the data structure **"dictionary" as a red-black tree** (the class **SortedDictionary<K,T>**) is a structure storing key-value pairs where keys are ordered increasingly (sorted). The structure provides a **fast execution** of basic operations (add an element, search by key and remove an element). The complexity of these operations is **logarithmic** – O(log(N)). Thus, it will take 10 steps for add / search / remove when the dictionary holds 1,000 elements and 20 steps in case of 1,000,000 elements.

Unlike hash-tables, where we can reach linear complexity if we pick a bad hash-function, in **SortedDictionary<K,T>** the count of the steps of the basic operations in the average and worst case are the same – $\mathbf{log_2(N)}$. When we work with balanced trees, there is no hashing, no collisions and no risk of using a bad hash-function.

Again, as in the hash-tables, one key can be stored at most once in the structure. If we want to associate several values with one key, we should use some kind of a list for the values, for example **List<T>**.

SortedDictionary<K,T> holds internally its values in a **red-black balanced tree** ordered by key. This means if we traverse the structure (using its iterator or **foreach** loop in C#) we will get the elements sorted in ascending order by key. Sometimes this can be very useful property.

Use **SortedDictionary<K,T>** when you need a structure which can add, search and remove an element fast and you also need to extract the elements sorted in ascending order. In general **Dictionary<K,T>** works a bit faster than **SortedDictionary<K,T>** and is preferable.

As an example of using a **SortedDictionary<K,T>**, we can give the following task: **find all the words in a text file, which occur exactly 10 times, and print them alphabetically**. This is a task that we can solve as successful with **Dictionary<K,T>** too, but we will have to do an additional sorting at the

end. For the solution of this task we can use **SortedDictionary<string, int>** and to traverse through all the words in the text file. For each word we will keep in the sorted dictionary how many times we have encountered it. After that we can go through all the elements in the dictionary and print those words, which have been encountered exactly 10 times. They will be alphabetically ordered, since this is the natural internal order of the sorted dictionary data structure.

Use SortedDictionary<K,T> when you want fast addition of elements and searching by key as well as the elements to be sorted by key.

Set, Implemented with a Hash-Table (HashSet<T>)

The data structure **"set" is a collection of elements with no duplicates**. The basic operations are adding an element to the set, checking if an element belongs to the set (searching) and removing an element from the set. The operation searching through index is not supported, i.e. we do not have a direct access to the elements via ordering number, because in this structure there is not any order.

Set, implemented with a **hash-table** (the class **HashSet<T>**) is a special case of a hash-table, in which we have only keys. The values associated with these keys do not matter.

As in the hash-table, the basic operations in the data structure **HashSet<T>** are implemented with a **constant complexity O(1)**. Another similarity to hash-table is if we choose a bad hash-function, we can reach a linear complexity executing the basic operations. Fortunately in practice this almost never happens.

As an example of using a **HashSet<T>**, we can point out the task of finding all the different words in a text file.

Use HashSet<T>, when you have to quickly add elements to a set and check whether a given element belongs to a set.

Set, Implemented with a Balanced Tree (SortedSet<T>)

The data structure set, implemented with a red-black tree, is a special case of **SortedDictionary<K,T>** in which keys and values coincide.

Similar to **SortedDictionary<K,T>**, the basic operations in **SortedSet<T>** are executed with logarithmic complexity **O(log(N))**, which is the same in the average and worst case.

As an example of using a **SortedSet<T>** we can point out the task of finding all the different words in a given text file and printing them alphabetically ordered.

Use SortedSet<T>, when you have to quickly add an element to a set and check whether given element belongs to the set as well as need all the elements sorted in ascending order.

Choosing a Data Structure – Examples

We are going to **show several problems**, where the choice of an appropriate data structure is crucial to the efficiency of their solution. The purpose of this is to show you typical situations, in which the reviewed data structures are used and to teach you in what scenarios what data structures you should use.

Generating Subsets

It is given a set of strings **S**. For example **S** = {ocean, beer, money, happiness}. The task is to write a program, which prints **all subsets of S**.

The problem has many and different solutions, but we are going to focus on the following one: We **start from the empty set** (with 0 elements):

```
{}
```

We **add to it every element of S** and we get a collection of subsets with one element:

```
{ocean}, {beer}, {money}, {happiness}
```

To each of the one-elemental subsets we **add every element from S**, which has not been added yet to the corresponding subset and now we have all two-elemental subsets:

```
{ocean, beer}, {ocean, money}, {ocen, happiness}, {beer, money},
{beer, happiness}, {money, happiness}
```

If we keep on the same way, we will get all 3-elemental subsets and after that all 4-elemental etc. to the N-elemental subsets.

How to implement this algorithm? We have to choose **appropriate data structures**, right?

We can start with the data structure keeping the initial set of elements **S**. It can be an **array, linked list, dynamic array** (**List<string>**) or **set**, implemented as **SortedSet<string>** or **HashSet<string>**. To answer the question which structure is the most appropriate, let's think of which are the operations we are going to do on this structure. We can think of only one operation –

traversing through all the elements of **S**. This operation can be implemented efficiently with any of these structures. We **choose an array** because it is the simplest data structure of all and it is easy to work with.

The next step is to **pick a structure** in which we will store **one of the subsets** we generate, for example {ocean, happiness}. Again we ask ourselves the question what are the operations we execute on this subset of words. The operations are a check whether an element exists and an addition of an element, right? Which data structure quickly implements both operations? The arrays and lists do not search quickly, dictionaries store key-value pairs, which is not our case. Almost no options are left, so we are going to see what the data structure set offers. It supports a quick searching and addition. Which implementation to choose – **SortedSet<string>** or **HashSet<string>**? We do not have a requirement to sort the words in alphabetical order, so we **choose the faster implementation – HashSet<string>**.

Lastly, we will choose one more data structure in which we are going to keep the **collection of the subsets** of words, for example:

```
{ocean, beer}, {sea, money}, {sea, happiness}, {beer, money},
{beer, happiness}, {money, happiness}
```

Using this structure we have to be able to add as well as traverse through all its elements consecutively. The following structures meet the requirements: **list, stack, queue and set**. In each of them we can add quickly and go through its elements. If we examine the algorithm for generating subsets, we will notice each is processed in style: "**first generated, first processed**". The subset, which had been firstly generated, has been firstly processed and subsets with one more element have been generated from it, right? Therefore our algorithm will most accurately fit the data structure **queue**. We can describe the algorithm as follows:

1. We start with a queue, containing the empty set **{}**.
2. We dequeue an element called **subset** and try to add each element from S which **subset** does not contain. The result is a set, which we enqueue.
3. We repeat step 2 until the queue becomes empty.

You can see how a few thoughts brought us to the classical algorithm "**breadth-first search**" (**BFS**). Once we know what data structures we should use, implementation is quick and easy. Here is how it might look:

```
string[] words = {"ocean", "beer", "money", "happiness"};
Queue<HashSet<string>> subsetsQueue =
  new Queue<HashSet<string>>();
HashSet<string> emptySet = new HashSet<string>();
subsetsQueue.Enqueue(emptySet);
while (subsetsQueue.Count > 0)
```

```
{
  HashSet<String> subset = subsetsQueue.Dequeue();

  // Print current subset
  Console.Write("{ ");
  foreach (string word in subset)
  {
    Console.Write("{0} ", word);
  }
  Console.WriteLine("}");

  // Generate and enqueue all possible child subsets
  foreach (string element in words)
  {
    if (! subset.Contains(element))
    {
      HashSet<string> newSubset = new HashSet<string>();
      newSubset.UnionWith(subset);
      newSubset.Add(element);
      subsetsQueue.Enqueue(newSubset);
    }
  }
}
```

If we execute the code above, we will see that it successfully **generates all subsets of S**, but some of them are **generated twice**.

```
{ }
{ ocean }
{ beer }
{ money }
{ happiness }
{ ocean beer }
{ ocean money }
{ ocean happiness }
{ beer ocean }
…
```

In the example the subsets **{ ocean beer }** and **{ beer ocean }** are actually one and the same subset. It seems we have not thought of duplicates, which occur when we mix the order of elements in the same subset. How can we avoid duplicates?

Let's associate the words by their indices.

ocean → 0

beer → 1
money → 2
happiness → 3

Since the subsets {1, 2, 3} and {2, 1, 3} are actually one and a same subset, in order to avoid duplicates, we are going to impose a requirement to generate only subsets, in which the **indices are in ascending order**. Instead of subsets of words we can keep subsets of indices, right? In these subsets of indices we need two operations: adding an index and getting the biggest index so we can add only indices bigger than it. Obviously we do not need **HashSet<T>** anymore, but we can successfully use **List<T>**, in which the elements are ordered in ascending order by index and the biggest element is naturally placed last.

Finally, our algorithm looks something like this:

1. Let **N** be the number of elements in **S**. We start with a **queue**, containing the **empty set {}**.

2. We dequeue an element called **subset**. Let **start** be the biggest index in **subset**. We add to **subset** all indices, which are bigger than **start** and smaller than N. As a result we get several new subsets, which we enqueue.

3. Repeat step 2 until the queue is empty.

Here is how the **implementation of the new algorithm** looks like:

```
using System;
using System.Collections.Generic;

public class Subsets
{
  static string[] words = { "ocean", "beer", "money",
    "happiness" };

  static void Main()
  {
    Queue<List<int>> subsetsQueue = new Queue<List<int>>();
    List<int> emptySet = new List<int>();
    subsetsQueue.Enqueue(emptySet);
    while (subsetsQueue.Count > 0)
    {
      List<int> subset = subsetsQueue.Dequeue();
      Print(subset);
      int start = -1;
      if (subset.Count > 0)
      {
        start = subset[subset.Count - 1];
```

```
            }
            for (int i = start + 1; i < words.Length; i++)
            {
                List<int> newSubset = new List<int>();
                newSubset.AddRange(subset);
                newSubset.Add(i);
                subsetsQueue.Enqueue(newSubset);
            }
        }
    }

    static void Print(List<int> subset) {
        Console.Write("[ ");
        for (int i=0; i<subset.Count; i++) {
            int index = subset[i];
            Console.Write("{0} ", words[index]);
        }
        Console.WriteLine("]");
    }
}
```

If we run the program we will get the following **correct result**:

```
[ ]
[ ocean ]
[ beer ]
[ money ]
[ happiness ]
[ ocean beer ]
[ ocean money ]
[ ocean happiness ]
[ beer money ]
[ beer happiness ]
[ money happiness ]
[ ocean beer money ]
[ ocean beer happiness ]
[ ocean money happiness ]
[ beer money happiness ]
[ ocean beer money happiness ]
```

Sorting Students

It is given a **text file**, containing the data of a group of **students and courses** which they attend, separated by |. The file looks like this:

```
Chris | Jones | C#
Mia | Smith | PHP
Chris | Jones | Java
Peter | Jones | C#
Sophia | Wilson | Java
Mia | Wilson | C#
Mia | Smith | C#
```

Write a program **printing all courses and the students**, who have joined them, **ordered by last name**, and then by first name (if the last names match).

We can implement the problem using a **hash-table**, which will hold a list of students by a course name. We are choosing a hash-table, because we can **quickly search by course name** in it.

In order to meet the requirements for an order by name and surname, we are going to **sort the particular list of students** from each course, before we print it. Another option is to use **SortedSet<T>** for the students attending each course (because it is internally sorted), but since one can have students with the same name, we have to use **SortedSet<List<String>>**. It becomes too complicated. We choose the easier way – using **List<Student>** and sorting it before we print it.

In any case we will have to implement the **IComparable** interface so we can define **the order of the elements** of type **Student** according to the task requirements. It is important to firstly compare the family names and if they are the same to compare the first names. We remind that in order to sort the elements of a given class it is explicitly necessary to define the logic of their order. In .NET Framework this is done by the **IComparable<T>** interface (or through lambda functions like we shall see in the chapter "Lambda Expressions and LINQ"). Let's define the class **Student** and implement **IComparable<Student>**. We get something like this:

```
public class Student : IComparable<Student>
{
  private string firstName;
  private string lastName;

  public Student(string firstName, string lastName)
  {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  public int CompareTo(Student student)
  {
```

```
        int result = lastName.CompareTo(student.lastName);
        if (result == 0)
        {
            result = firstName.CompareTo(student.firstName);
        }
        return result;
    }

    public override String ToString()
    {
        return firstName + " " + lastName;
    }
}
```

Now we are able to write the code, which **reads the students and their courses and stores them in a hash-table**, which keeps a list of students by a course name (**Dictionary<string, List<Student>>**). And then it is easy – we iterate over the courses, sort the students and print them:

```
// Read the file and build the hash-table of courses
Dictionary<string, List<Student>> courses =
    new Dictionary<string, List<Student>>();
StreamReader reader = new StreamReader("Students.txt");
using (reader)
{
    while (true)
    {
        string line = reader.ReadLine();
        if (line == null)
        {
            break;
        }
        string[] entry = line.Split(new char[] { '|' });
        string firstName = entry[0].Trim();
        string lastName = entry[1].Trim();
        string course = entry[2].Trim();
        List<Student> students;
        if (! courses.TryGetValue(course, out students))
        {
            // New course -> create a list of students for it
            students = new List<Student>();
            courses.Add(course, students);
        }
        Student student = new Student(firstName, lastName);
        students.Add(student);
```

```
    }
}

// Print the courses and their students
foreach (string course in courses.Keys)
{
    Console.WriteLine("Course " + course + ":");
    List<Student> students = courses[course];
    students.Sort();
    foreach (Student student in students)
    {
        Console.WriteLine("\t{0}", student);
    }
}
```

The above code first **parses all the lines** consecutively by splitting them by a vertical bar "|". Secondly it cleans the spaces from the beginning and the end. After storing every student's information, it is **checked in the hash-table** whether its course exists. If the course has been found, the student is added to the list of the students of this course. Otherwise, a new list is created and the student is added to it. Then the list is added in the hash-table using the course name as a key.

Printing the courses and students is not difficult. All keys are extracted from the hash-table. These are the names of the courses. For each course its students' list is **extracted, sorted and printed**. The sorting is made by the built-in method **Sort()** using the comparison method **CompareTo(…)** from the interface **IComparable<T>** as defined in the class **Student** (comparison firstly by family name, and if they are the same, comparison by first name). At the end the sorted students are printed by the overridden virtual method **ToString()**. Here is how the output of the upper program looks:

```
Course C#:
        Chris Jones
        Peter Jones
        Mia Smith
        Mia Wilson
Course PHP:
        Mia Smith
Course Java:
        Chris Jones
        Sophia Wilson
```

Sorting a Phone Book

It is given a **text file, containing people's names, their city names and phone numbers**. The file looks like this:

```
Kenneth | Virginia Beach | 1-541-754-3010
Paul    | San Antonio    | 1-535-675-6745
Mary    | Portland       | 1-234-765-1983
Laura   | San Antonio    | 1-454-345-2345
Donna   | Virginia Beach | 1-387-387-2389
```

Write a program which prints all the **city names in an alphabetical order** and for each one of them prints all **people's names in alphabetical order** and their corresponding **phone number**.

The problem can be solved in many ways, for example we sort by two criteria: firstly by city name and secondly by person name and then we print the phone book.

However, let's solve the problem without sorting, but by using the standard data structures in .NET Framework. We want the city names to be sorted. This means that it is best to use a data structure, which internally keeps the elements **sorted**. Such as, for example, a balanced search tree – **SortedSet<T>** or **SortedDictionary<K,T>**. Since every record from the phone book contains beside a city name – other data, it is more convenient to have a **SortedDictionary<K,T>**, which keeps a list of people's names and their phone numbers. We want the list of the people's names from every city to be sorted in alphabetical order by name. Hence, we can use the data structure **SortedDictionary<K,T>** again. The key will be the name of the person and its value will be his phone number.

At the end we get the nested structure **SortedDictionary<string, SortedDictionary<string, string>>**. Here is a sample implementation, which shows how we can solve the problem using this structure:

```
// Read the file and build the phone book
SortedDictionary<string, SortedDictionary<string, string>>
  phonesByTown = new SortedDictionary<string,
    SortedDictionary<string, string>>();
StreamReader reader = new StreamReader("PhoneBook.txt");
using (reader)
{
  while (true)
  {
    string line = reader.ReadLine();
    if (line == null)
    {
      break;
```

```
        }
        string[] entry = line.Split(new char[]{'|'});
        string name = entry[0].Trim();
        string town = entry[1].Trim();
        string phone = entry[2].Trim();

        SortedDictionary<string, string> phoneBook;
        if (! phonesByTown.TryGetValue(town, out phoneBook))
        {
            // This town is new. Create a phone book for it
            phoneBook = new SortedDictionary<string, string>();
            phonesByTown.Add(town, phoneBook);
        }
        phoneBook.Add(name, phone);
    }
}

// Print the phone book by towns
foreach (string town in phonesByTown.Keys)
{
    Console.WriteLine("Town " + town + ":");
    SortedDictionary<string, string> phoneBook =
        phonesByTown[town];
    foreach (var entry in phoneBook)
    {
        string name = entry.Key;
        string phone = entry.Value;
        Console.WriteLine("\t{0} - {1}", name, phone);
    }
}
```

If we execute this sample code with input – the sample phone book, we will get the expected result:

```
Town Portland:
        Mary - 1-234-765-1983
Town San Antonio:
        Laura - 1-454-345-2345
        Paul - 1-535-675-6745
Town Virginia Beach:
        Donna - 1-387-387-2389
        Kenneth - 1-541-754-3010
```

Searching in a Phone Book

Here is another problem, so we can strengthen the way, in which we think in order to **choose appropriate data structures**. A **phone book** is stored in a text file, containing **names of people, their city names and phone numbers**. People's names can be in the format first name or nickname or first name + last name or first name + surname + last name. The file could have the following look:

```
Kevin Clark       | Virginia beach | 1-454-345-2345
Skiller           | San Antonio    | 1-566-533-2789
Kevin Clark Jones | Portland       | 1-432-556-6533
Linda Johnson     | San Antonio    | 1-123-345-2456
Kevin             | Phoenix        | 1-564-254-4352
Kevin Garcia      | Virginia Beach | 1-445-456-6732
Kevin             | Phoenix        | 1-134-654-7424
```

It is possible several people to be given under the same name or even the same city. It is possible someone to have **several phone numbers**. In this case he is given several times in the input file. **Phone book could be huge** (up to 1,000,000 records).

A file holding a sequence of queries is given. The **queries** are two types:

- **Search by name / nickname / surname / last name**. The query looks like this: **`list(name)`**.
- **Search by name / nickname / surname / last name + city name**. The query looks like this: **`find(name, town)`**.

Here is a sample query file:

```
list(Kevin)
find(Angel, San Antonio)
list(Linda)
list(Clark)
find(Jones, Phoenix)
list(Grandma)
```

Write a program, which by given phone book and query file **executes and respond to all the queries**. For each query a list of records in the phone book has to be printed or the message "**Not found**", if the query cannot find anything. **Queries could be a large number**, for example 50,000.

This problem is not as easy as the previous ones. One easy to implement solution could be to **scan the entire phone book** for every query and extract all records in which there is a match with the searched information. But this **will work slowly**, because the records and queries could be a lot. It is necessary to find a way for a quick search without scanning the entire phone book every time.

In paper phone books the numbers are given by the **people's name, sorted in alphabetical order**. Sorting will not help us, because someone can search by first name, other – by last name, third – by nickname and city name. We have to search by any of the above at the same time. The question is **how do we do it**?

If we think a bit it will occur to us that the problem requires **searching by any of the words**, which can be seen at the first column of the phone book and eventually by the combination of a word from the first column and a town from the second one. We know that the fastest search is implemented with a **hash-table**. So far so good, but what are we going to keep as a key and value in the hash-table?

What if we use **several hash-tables**: one for searching a word from the first column, another for searching in the second column, third for searching by city and so on? If we think a bit more, we will ask ourselves the question – why do we need several hash-tables? Can't we **search in one common hash-table**? If we have a name "Peter Jones", we will store his phone number under the keys "Peter" and "Jones". If someone searches by any of these keys, he will find Peter's phone number.

So far so good, but how do we **search by both first name and city name**, for example "Peter from Virginia Beach"? It is possible firstly to find all with a name "Peter" and then to print those who are from Virginia Beach. This will work, but if there are a lot of people named Peter, searching will work slowly. Then why don't we make a **hash-table with a key name of a person and value another hash-table**, which by city name will return a list of phone numbers? This could work. We have done something similar in the previous task, haven't we?

Can we come up with **something smarter**? Can't we just put the phone numbers of all the people named Peter from Virginia Beach under a key "**Peter from Virginia Beach**" in the main hash-table in the phone book? It seems this could solve our problem and we will use only one hash-table for all the queries.

Using the last idea, we could invent the following **algorithm**: we read line by line from the phone book and **for each word from the name of a person** d_1, d_2, …, d_k and **for each city name t** we **make new records in the phonebook hash-table** by the following keys: d_1, d_2, …, d_k, "d_1 from t", "d_2 from t", …, "d_k from t". Now it is guaranteed we could search by any of a person's names as well as name and town. In order to search without bothering about letter case we can transform the words to lowercase in advance. After that the searching is trivial – we just search in the hash-table by a given word **d** or if a town **t** is given "**d from t**". Since we could have many phone numbers under the same key, for a value in the hash-table we should use a list of strings (`List<string>`).

Let's skim through an implementation of the described algorithm:

```
class PhoneBookFinder
{
  const string PhoneBookFileName = "PhoneBook.txt";
  const string QueriesFileName = "Queries.txt";

  static Dictionary<string, List<string>> phoneBook =
    new Dictionary<string, List<string>>();

  static void Main()
  {
    ReadPhoneBook();
    ProcessQueries();
  }

  static void ReadPhoneBook()
  {
    StreamReader reader = new StreamReader(PhoneBookFileName);
    using (reader)
    {
      while (true)
      {
        string line = reader.ReadLine();
        if (line == null)
        {
          break;
        }
        string[] entry = line.Split(new char[]{'|'});
        string names = entry[0].Trim();
        string town = entry[1].Trim();

        string[] nameTokens =
          names.Split(new char[] {' ', '\t'} );
        foreach (string name in nameTokens)
        {
          AddToPhoneBook(name, line);
          string nameAndTown = CombineNameAndTown(town, name);
          AddToPhoneBook(nameAndTown, line);
        }
      }
    }
  }

  static string CombineNameAndTown(  string town, string name)
  {
```

```
        return name + " from " + town;
    }

    static void AddToPhoneBook(string name, string entry)
    {
        name = name.ToLower();
        List<string> entries;
        if (! phoneBook.TryGetValue(name, out entries))
        {
            entries = new List<string>();
            phoneBook.Add(name, entries);
        }
        entries.Add(entry);
    }

    static void ProcessQueries()
    {
        StreamReader reader = new StreamReader(QueriesFileName);
        using (reader)
        {
            while (true)
            {
                string query = reader.ReadLine();
                if (query == null)
                {
                    break;
                }
                ProcessQuery(query);
            }
        }
    }

    static void ProcessQuery(string query)
    {
        if (query.StartsWith("list("))
        {
            int listLen = "list(".Length;
            string name = query.Substring(
                listLen, query.Length-listLen-1);
            name = name.Trim().ToLower();
            PrintAllMatches(name);
        }
        else if (query.StartsWith("find("))
        {
```

```
            string[] queryParams = query.Split(
                new char[] { '(', ' ', ',', ')' },
                StringSplitOptions.RemoveEmptyEntries);
            string name = queryParams[1];
            name = name.Trim().ToLower();
            string town = queryParams[2];
            town = town.Trim().ToLower();
            string nameAndTown =
                CombineNameAndTown(town, name);
            PrintAllMatches(nameAndTown);
        }
        else
        {
            Console.WriteLine(
                query + " is invalid command!");
        }
    }

    static void PrintAllMatches(string key)
    {
        List<string> allMatches;
        if (phoneBook.TryGetValue(key, out allMatches))
        {
            foreach (string entry in allMatches)
            {
                Console.WriteLine(entry);
            }
        }
        else
        {
            Console.WriteLine("Not found!");
        }
        Console.WriteLine();
    }
}
```

While reading the phone book line by line and splitting by the vertical bar "|", we **extract the three columns** (names, city name and phone number). After that the names are split and each word is **added in the hash-table**. Additionally we add each word, combined with the city name (so that we can search by name + city name).

The second part of the algorithm is the **command execution**. In this part each line from the query file is read and processed. The process includes parsing the command, extracting the name or name and city name and

searching. The search is directly done by using the **hash-table**, which is created after reading the phone book file.

To be able to ignore the difference between lowercase and uppercase, all keys in the hash-table are added as lowercase. When we search, we do it lowercase too.

Choosing a Data Structure – Conclusions

By the many examples it is clear that the choice of an appropriate data structure is **highly dependable on the specific task**. Sometimes **data structures have to be combined** or we have to use several of them simultaneously.

What data structure should we pick mostly **depends on the operations we will perform**, so always ask yourselves "what operations should the structure, I need, perform efficiently". If you are familiar with the operations, you can easily conform which structure does them most efficiently and at the same time is easy and handy.

In order to efficiently choose an appropriate data structure, you should firstly **invent the algorithm**, which you are going to implement, and then **look for an appropriate data structures** for it.

Always go from the algorithm to the data structures, never backwards.

External Libraries with .NET Collections

It is a well-known fact that the standard data structures in .NET Framework **System.Collections.Generic** have pretty poor functionality. It lacks implementations of basic concepts in data structures such as multi-sets, priority queues, for which there should be standard classes as well as basic system interfaces.

When we have to use a **special data structure**, which is not standardly implemented in .NET Framework, we have two options:

- First option: we **implement the data structure ourselves**. This gives us flexibility, because the implementation will completely meet our needs, but it takes a lot of time and it has a great chance of making mistakes. For example, if one has to qualitatively implement a balanced tree, this may take an experienced software developer several days (along with the tests). If the same is implemented by inexperienced software developer it will take a lot more time and most probably there will be errors in the implementation.
- Second option (generally preferable): **find an external library**, which has a full implementation of the needed functionality. This approach has an advantage of saving us time and troubles, because in most cases the external libraries of data structures are well-tested. They have been

used for years by thousands of software developers and this makes them mature and reliable.

Power Collections for .NET

One of the most popular and richest libraries with efficient implementations of the fundamental data structures for C# and .NET software developers is the open-source project "**Wintellect's Power Collections for .NET**" – http://powercollections.codeplex.com. It provides free, reliable, efficient, fast and handy implementations of the following commonly used **data structures**, which are missing or partly-implemented in .NET framework:

- **Set<T>** – **set** of elements, **implemented with a hash-table**. It efficiently implements the basic operations over sets: adding, deleting and searching an element as well as union, intersection, difference between sets and many more. By functionality and way of work the class looks like the standard class **HashSet<T>** in .NET Framework.

- **Bag<T>** – **multi-set of elements** (set with duplicates), implemented with a **hash-table**. It efficiently implements all basic operations over multi-sets.

- **OrderedSet<T>** – **ordered set of elements** (without duplicates), implemented with a **balanced search tree**. It efficiently implements all basic operations over sets and when traversing through its elements it returns them in ascending order (according to the used comparer). It allows a fast extraction of subsets of values in a given interval.

- **OrderedBag<T>** – **ordered multi-set** of elements, implemented with a **balanced search tree**. It efficiently implements all basic operations over multi-sets and when going through all its elements it returns them in ascending order (according to the used comparer). It allows a quick extraction of subsets of values in a given interval.

- **MultiDictionary<K,T>** – it is a **hash-table allowing key duplicates**. For every key there is a collection of values stored, not one single value.

- **OrderedDictionary<K,T>** – it represents a **dictionary, implemented with a balanced search tree**. It allows a fast search by key and when going through its elements it returns them in ascending order. It enables us to quickly extract the values from a given key range. By functionality and way of work the class looks like the standard class **SortedDictionary<K,T>** in .NET Framework.

- **Deque<T>** – represents efficient implementation of a queue with two ends (**double ended queue**), which practically combines the data structures stack and queue. It allows efficient addition, extraction and deletion of elements in both ends.

- **BagList<T>** – **list of elements, accessed through index**, which allows a **quick insertion and deletion** of an element from a particular position. The operations index accessing, adding, inserting at position

and removing an element from position have a complexity O(log N). The implementtation is with a balanced tree. The structure is a good alternative of **List<T>**, in which the insertion and removal of element at a particular position takes linear time because of the need of the replacement of linear number of elements to the left or right.

We let the reader the opportunity to download the library "**Power Collections for .NET**" from its site and to **experiment with it**. It can be very useful when you solve some of the problems from the exercises.

C5 Collections for .NET

Another very powerful library of data structures and collection classes is "**The C5 Generic Collection Library for C# and CLI**" (www.itu.dk/research/c5/). It provides standard interfaces and collection classes like **lists**, **sets**, **bags**, **multi-sets**, **balanced trees** and **hash tables**, as well as **non-traditional data structures** like "hashed linked list", "wrapped arrays" and "interval heaps". It also describes a set of collection-related **algorithms** and **patterns**, such as "read-only access", "random selection", "removing duplicates", etc. The library comes with solid documentation (a book of 250 pages). The C5 collections and the book about them are the ultimate resource for data structure developers.

Exercises

1. Hash-tables do not allow storing **more than one value in a key**. How can we get around this restriction? Define a class to hold multiple values in a hash-table.

2. Implement a data structure, which can quickly do the following two operations: **add an element** and **extract the smallest element**. The structure should accept adding duplicated elements.

3. It is given a text file **students.txt** containing information about students and their specialty in the following format:

```
Steven Davis | Computer Science
Joseph Johnson | Software Engeneering
Helen Mitchell | Public Relations
Nicolas Carter | Computer Science
Susan Green | Public Relations
William Johnson | Software Engeneering
```

   Using **SortedDictionary<K,T>** print on the console the specialties in an alphabetical order and for each of them print the names of the students, firstly sorted by family name and secondly – by first name, as shown:

```
Computer Sciences: Nicolas Carter, Steven Davis
Public Relations: Susan Green, Helen Mitchell
```

```
Software Engeneering: Joseph Johnson, William Johnson
```

4. Implement a class **BiDictionary<K1,K2,T>,** which allows adding triplets **{key1, key2, value}** and quickly search by either of the keys **key1**, **key2** as well as searching by combination of the both keys. Note: Adding many elements with the same keys is allowed.

5. A big chain of supermarkets sell **millions of products**. Each of them has a unique number (barcode), producer, name and price. What data structure could we use in order to quickly **find all products, which cost between 5 and 10 dollars**?

6. A **timetable** of a conference hall is a list of events in a format **[starting date and time; ending date and time; event's name]**. What data structure could we to be able to quickly **add events** and **quickly check whether the hall is available in a given interval** [starting date and time; ending date and time]?

7. Implement the data structure **PriorityQueue<T>**, which offers quick execution of the following operations: **adding an element**, **extracting the smallest element**.

8. Imagine you **develop a search engine**, which gathers all the advertisements for used cars in ten websites for the last few years. After that the search engine allows a quick search by one or several criteria: a brand, model, color, year of production and price. You are not allowed to use database management system (like SQL Server, MySQL or MongoDB) and you must implement your own indexing in the memory, without storing it to the hard disk and without using LINQ. When one searches by price minimal and maximal price is given. When one searches by year of production a starting and ending years are given. What data structures would you use in order to ensure fast searching by one or several criteria?

Solutions and Guidelines

1. You can use **Dictionary<key, List<value>>** or create your own class **ValuesCollection**, which can take care of the values with the same key and use **Dictionary<key, ValuesCollection>**.

2. You can use **SortedSet<List<int>>** and its operations **Add()** and **First()**. **SortedSet<T>** keeps the elements in it sorted and can accept external **IComparer<T>**.

   The problem has a more efficient solution though – the data structure called "**binary heap**". You can read about it on Wikipedia: http://en.wikipedia.org/wiki/Binary_heap.

3. The task is similar to the one from the section "Sorting Students".

4. One of the solutions to this task is to use two instances of the class **Dictionary<K,T>** for each of the two keys and when you add or remove

an element from **BiDictionary<K1,K2,T>**, you add or remove the element from the **two hash-tables** correspondingly. When you search by first or second key, you should check the elements in the first or the second hash-table respectively. When you search by two keys, you could search in the two hash-tables separately and intersect the matching subsets.

Another, simpler approach is to hold 3 hash tables: **Dictionary<K1,T>**, **Dictionary<K2,T>** and **Dictionary<Tuple<K1,K2>,T>**. The system generic class **Tuple<K1,K2>** can be used to combine two keys and use it as a **composite key**.

5. If we keep the products sorted by price in an **array** (for example in **List<Product>**, which we firstly fill and then **sort**), in order to find all the products which cost between 5 and 10 bucks we can use a **binary search** twice. Firstly we can find the smallest index **start**, in which lies a product costing at least 5 bucks. After that we can find the biggest index **end**, in which lies a product costing at most 10 bucks. All the products at positions in the interval [**start** … **end**] will cost between 5 and 10 dollars. If you are interested in the algorithm binary search in a sorted array you could inform yourself reading Wikipedia: http://en.wikipedia.org/wiki/Binary_search.

   Generally the approach using a sorted array and binary search in it works excellent, but there is a disadvantage: the addition in a sorted array is a very slow operation, because it requires moving a linear number of elements with one position ahead of the inserted new element.

   To overcome this we can use the class **SortedSet<T>**. It supports **fast insertion** keeping the elements in a **sorted order**. It has an operation **SortedSet<T>.GetViewBetween(lowerBound, upperBound)** that returns a subset of the elements in certain **range** (interval).

   You may also use the class **OrderedSet<T>** from "Wintellect's Power Collections for .NET" library (http://www.codeplex.com/PowerCollections) which is more powerful and more flexible. It has a method for extracting a sub-range of values: **OrderedSet<T>.Range(from, fromInclusive, to, toInclusive)**.

6. We can create **two sorted arrays** (**List<Event>**): the first will keep the events sorted in ascending order by **starting date and time**; the second will keep the same events sorted by **ending date and time**. By using binary search we can find all the events which can be partly or fully found between the two moments of time [**start, end**] by doing the following:

   - Find the **set S** of all events starting after the moment **start** (using binary search).

   - We can find all the **set E** of all events ending before the moment **end** (using binary search).

   - **Intersect** these two sets: **C = S ∩ E**. If the intersection **S** of the two sets of events have common elements (**S** in non-empty set), then in

the searched interval [**start** … **end**] the hall is occupied. Otherwise it is available.

This solution has a **disadvantage**: adding elements in the sorted arrays will be slow. We should either add all elements initially and then sort the two arrays and never change them afterwards or try to keep the arrays sorted when adding new elements (which will be slow).

Another solution, which is **easier** to implement and **more efficient**, is to use two instances of the class **OrderedBag<T>** from the "Power Collections for .NET" library (the first with event's **start** date and time as a key and the second with event's **end** date and time as a key). The class has methods to extract the subsets **S** and **E**: **RangeFrom(from, fromInclusive)** and **RangeTo(to, toInclusive)**. We still will need to intersect these sets and check whether their intersection is empty or not.

The most efficient solution is to use a data structure called "**interval tree**". Read more in Wikipedia: http://en.wikipedia.org/wiki/Interval_tree. You may find an **open source C# interval tree** implementation in CodePlex: http://intervaltree.codeplex.com.

7. Since there is no internal implementation of the data structure "**priority queue**" in .NET, you can use the data structure **OrderedBag<T>** from Wintellect's Power Collections. It had **Add(…)** and **GetFirst()** and **RemoveFirst()** methods. You can read more about priority queues on Wikipedia: http://en.wikipedia.org/wiki/Priority_Queue.

   The classic, **simplest efficient priority queue** implementation the data structure "**binary heap**": http://en.wikipedia.org/wiki/Binary_heap.

   An efficient ready-to-use C# implementation of priority queue is the class **IntervalHeap<T>** in the C5 Collections: http://www.itu.dk/research/c5/.

8. For **searching by brand, model and color** we can use one hash-table per each, which will search by a given criteria and return a list of cars (**Dictionary<string, List<Car>>**).

   For searching by **year of production** and **price range** we can use lists **List<Car>**, sorted in ascending order (and binary search).

   To search by several criteria at once we can extract the cars' **subsets by the first criteria**, after that the cars' **subsets by the second criteria** and so on. At the end we can find the **intersection** of the sets. Intersection of two sets can be found by looking for every element in the smaller set in the bigger set. The easiest way is **Car** to implement **Equals()** and **GetHashCode()** and after that to use the class **HashSet<Car>** for set intersections.

Chapter 20. Object-Oriented Programming Principles

In This Chapter

In this chapter we will familiarize ourselves with the **principles of object-oriented programming**: class **inheritance**, **interface** implementation, **abstraction** of data and behavior, **encapsulation** of data and class implementation, **polymorphism** and **virtual methods**. We will explain in details the principles of **cohesion** and **coupling**. We will briefly outline object-oriented modeling and how to create an object model based on a specific business problem. We will familiarize ourselves with **UML** and its role in object-oriented modeling. Finally, we will briefly discuss **design patterns** and illustrate some of those that are widely used in practice.

Let's Review: Classes and Objects

We introduced classes and objects in the chapter "Creating and Using Objects". Let's shortly review them again.

Classes are a description (**model**) of real objects and events referred to as entities. An example would be a class called "Student".

Classes possess **characteristics** – in programming they are referred to as **properties**. An example would be a set of grades.

Classes also expose **behavior** known in programming as **methods**. An example would be sitting an exam.

Methods and properties can be **visible** only within the scope of the class, which declared them and their descendants (**`private`** / **`protected`**), or visible to all other classes (**`public`**).

Objects are **instances** of classes. For example, John is a Student and Peter is also a Student.

Object-Oriented Programming (OOP)

Object-oriented programming is the successor of procedural (structural) programming. **Procedural programming** describes programs as groups of reusable code units (procedures) which define input and output parameters. Procedural programs consist of **procedures**, which invoke each other.

The **problem** with procedural programming is that **code reusability is hard** and limited – only procedures can be reused and it is hard to make them generic and flexible. There is no easy way to work with abstract data structures with different implementations.

The object-oriented approach relies on the paradigm that each and every program works with data that describes **entities** (objects or events) from real life. For example: accounting software systems work with invoices, items, warehouses, availabilities, sale orders, etc.

This is how objects came to be. They describe characteristics (properties) and behavior (methods) of such **real life entities**.

The main advantages and goals of OOP are to make complex software faster to develop and easier to maintain. OOP enables the easy reuse of code by applying simple and widely accepted rules (principles). Let's check them out.

Fundamental Principles of OOP

In order for a programming language to be **object-oriented**, it has to enable working with **classes** and **objects** as well as the implementation and use of the fundamental object-oriented principles and concepts: inheritance, abstraction, encapsulation and polymorphism. Let's summarize each of these **fundamental principles of OOP**:

- **Encapsulation**

  We will learn to **hide unnecessary details** in our classes and provide a clear and simple interface for working with them.

- **Inheritance**

  We will explain how **class hierarchies** improve code readability and enable the reuse of functionality.

- **Abstraction**

  We will learn how to **work through abstractions**: to deal with objects considering their important characteristics and ignore all other details.

- **Polymorphism**

  We will explain how to work in the same manner with different objects, which define a specific implementation of some **abstract behavior**.

Some OOP theorists also put the concept of **exception handling** as additional **fifth fundamental principle of OOP**. We shall not get into a detailed dispute about whether or not exceptions are part of OOP and rather will note that **exceptions are supported in all modern object-oriented languages** and are the primary mechanism of handling errors and unusual situations in object-oriented programming. **Exceptions always come together with OOP** and their importance is explained in details in the chapter "Exception Handling".

Inheritance

Inheritance is a fundamental principle of object-oriented programming. It allows a class to "inherit" (behavior or characteristics) of another, more general class. For example, a lion belongs to the biological family of cats (Felidae). All cats that have four paws, are predators and hunt their prey. This functionality can be coded once in the **Felidae** class and all its predators can reuse it – **Tiger**, **Puma**, **Bobcat**, etc. Inheritance is described as **is-kind-of relationship**, e.g. **Tiger** is kind of **Animal**.

How Does Inheritance Work in .NET?

Inheritance in .NET is defined with a special construct in the class declaration. In .NET and other modern programming languages, a class can inherit from a single class only **(single inheritance)**, unlike C++ which supports inheriting from multiple classes **(multiple inheritance)**. This limitation is necessitated by the difficulty in deciding which method to use when there are duplicate methods across classes (in C++, this problem is solved in a very complicated manner). In .NET, classes can inherit multiple interfaces, which we will discuss later.

The class from which we inherit is referred to as **parent class** or **base class / super class**.

Inheritance of Classes – Example

Let's take a look at an example of class inheritance in .NET. This is how a base class looks like:

Felidae.cs

```
/// <summary>Felidae is latin for "cats"</summary>
public class Felidae
{
  private bool male;

  // This constructor calls another constructor
  public Felidae() : this(true)
  {}

  // This is the constructor that is inherited
  public Felidae(bool male)
  {
    this.male = male;
  }

  public bool Male
  {
```

```
        get { return male; }
        set { this.male = value; }
    }
}
```

This is how the inheriting class, **Lion**, looks like:

Lion.cs

```
public class Lion : Felidae
{
    private int weight;

    // Keyword "base" will be explained in the next paragraph
    public Lion(bool male, int weight) : base(male)
    {
        this.weight = weight;
    }

    public int Weight
    {
        get { return weight; }
        set { this.weight = value; }
    }
}
```

The "base" Keyword

In the above example, we used the **keyword base** in the constructor of the class **Lion**. The keyword indicates that **the base class** must be used and allows access to its methods, constructors and member variables. Using **base()**, we can call the constructor of the base class. Using **base.Method(…)** we can invoke a method of the base class, pass parameters to it and use its results. Using **base.field** we can get the value of a member variable from the base class or assign a different one to it.

In .NET, methods inherited from the base class and declared as **virtual** can be **overridden**. This means **changing their implementation**; the original source code from the base class is ignored and new code takes its place. More on overriding methods we will discuss in "Virtual Methods".

We can invoke non-overridden methods from the base class without using the keyword **base**. Using the keyword is required only if we have an overridden method or variable with the same name in the inheriting class.

The keyword `base` can be used explicitly for clarity. `base.method(…)` calls a method, which is necessarily from the base class. Such source code is easier to read, because we know where to look for the method in question.

Bear in mind that using the keyword `this` is not the same. It can mean accessing a method from the current, as well as the base class.

You can take a look at the example in the section about access modifiers and inheritance. There it is clearly explained which members of the base class (methods, constructors and member variables) are **accessible**.

Constructors with Inheritance

When inheriting a class, our constructors must call the base class constructor, so that it can **initialize its member variables**. If we do not do this explicitly, the compiler will place a call to the parameterless base class constructor, "**`:base()`**", at the beginning of all our inheriting class' constructors. Here is an example:

```
public class ExtendingClass : BaseClass
{
  public ExtendingClass() { … }
}
```

This actually looks like this (spot the differences):

```
public class ExtendingClass : BaseClass
{
  public ExtendingClass() : base() { … }
}
```

If the base class has no default constructor (one without parameters) or that constructor is hidden, our constructors need to explicitly call one of the other base class constructors. The omission of such a call will result in a compile-time error.

If a class has private constructors only, then it cannot be inherited.

If a class has private constructors only, then this could indicate many other things. For example, no-one (other than that class itself) can create instances of such a class. Actually, that's how one of the most popular design patterns (Singleton) is implemented.

The Singleton design pattern is described in details at the end of this chapter.

Constructors and the Keyword "base" – Example

Take a look at the **Lion** class from our last example. It does not have a default constructor. Let's examine a class inheriting from **Lion**:

AfricanLion.cs

```
public class AfricanLion : Lion
{
  // …

  // If we comment out the ": base(male, weight)" line
  // the class will not compile. Try it.
  public AfricanLion(bool male, int weight)
    : base(male, weight)
  {}

  public override string ToString()
  {
    return string.Format(
      "(AfricanLion, male: {0}, weight: {1})",
      this.Male, this.Weight);
  }

  // …
}
```

If we comment out the line "**:base(male, weight);**", the class **AfricanLion** will not compile. Try it.

Calling the constructor of a base class happens outside the body of the constructor. The idea is that the fields of the base class should be initialized before we start initializing fields of the inheriting class, because they might depend on a base class field.

Access Modifiers of Class Members and Inheritance

Let's review: in the "Defining Classes" chapter, we examined the basic **access modifiers**. Regarding members of a class (methods, properties and member variables) we examined the modifiers **public**, **private** and **internal**. Actually, there are two other modifiers: **protected** and **protected internal**. This is what they mean:

- **protected** defines class members which are not visible to users of the class (those who initialize and use it), but are **visible to all inheriting classes** (descendants).

- **protected internal** defines class members which are both **internal**, i.e. visible within the entire assembly, and **protected**, i.e. not visible outside the assembly, but visible to classes who inherit it (even outside the assembly).

When a base class is inherited:

- All of its **public**, **protected** and **protected internal** members (methods, properties, etc.) are **visible** to the inheriting class.
- All of its **private** methods, properties and member-variables are **not visible** to the inheriting class.
- All of its **internal** members are visible to the inheriting class, only if the base class and the inheriting class are **in the same assembly** (the same Visual Studio project).

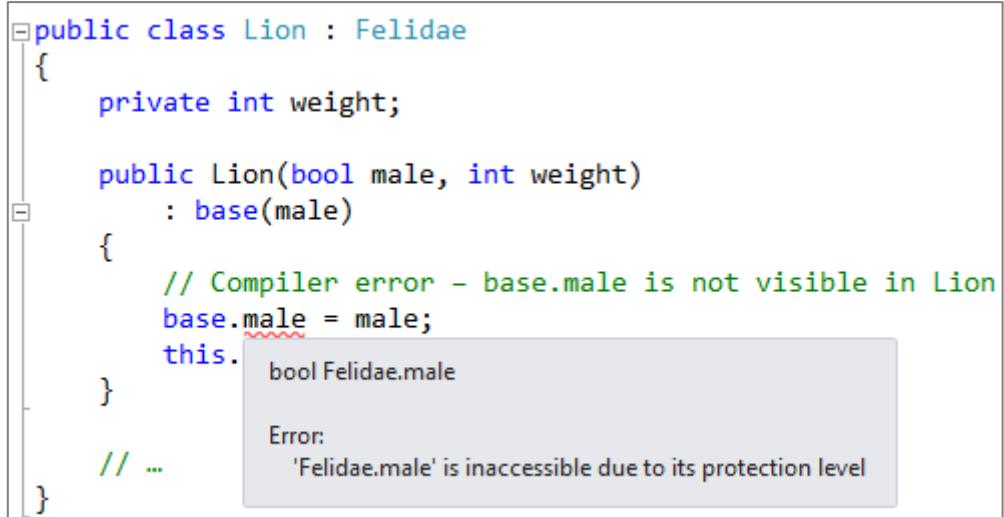
Here is an example, which demonstrates the **levels of visibility** with inheritance:

Felidae.cs

```
/// <summary>Latin for "cats"</summary>
public class Felidae
{
  private bool male;

  public Felidae() : this(true) {}

  public Felidae(bool male)
  {
    this.male = male;
  }

  public bool Male
  {
    get { return male; }
    set { this.male = value; }
  }
}
```

And this is how the class **Lion** looks like:

Lion.cs

```
public class Lion : Felidae
{
  private int weight;
```

```
    public Lion(bool male, int weight)
        : base(male)
    {
        // Compiler error – base.male is not visible in Lion
        base.male = male;
        this.weight = weight;
    }

    // …
}
```

If we try to compile this example, we will get an **error message**, because the **private** variable **male** in the class **Felidae** is not accessible to the class **Lion**:

```
public class Lion : Felidae
{
    private int weight;

    public Lion(bool male, int weight)
        : base(male)
    {
        // Compiler error – base.male is not visible in Lion
        base.male = male;
        this.
    }

    // …
}
```

```
bool Felidae.male

Error:
  'Felidae.male' is inaccessible due to its protection level
```

The System.Object Class

Object-oriented programming practically became popular with **C++**. In this language, it often becomes necessary to code classes, which must work with objects of any type. C++ solves this problem in a way that is not considered strictly object-oriented (by using **void** pointers).

The architects of .NET take a different approach. They create a class, which **all other classes inherit** (directly or indirectly). All objects can be perceived as instances of this class. It is convenient that this class contains important methods and their default implementation. This class is called **Object** (which is the same as **object** and **System.Object**).

In .NET **every class**, which does not inherit a class explicitly, **inherits the system class System.Object** by default. The compiler takes care of that.

Every class, which inherits from another class indirectly, inherits **Object** from it. This way every class inherits explicitly or implicitly from **Object** and contains all of its fields and methods.

Because of this property, **every class instance can be cast to Object**. A typical example of the advantages of implicit inheritance is its use with data structures, which we saw in the chapters on data structures. Untyped list structures (like **System.Collections.ArrayList**) can hold all kinds of objects, because they treat them as instances of the class **Object**.

The generic types (generics) have been provided specifically for working with collections and objects of different types (generics are further discussed in the chapter "Defining Classes"). They allow creating typified classes, e.g. a collection which works only with objects of type Lion.

.NET, Standard Libraries and Object

In .NET, there are a lot of predefined classes (we already covered a lot of them in the chapters on collections, text files and strings). These classes are part of the .NET framework; they are available wherever .NET is supported. These classes are referred to as **Common Type System (CTS)**.

.NET is one of the first frameworks, which provide such an extensive set of predefined classes. A lot of them work with **Object** so that they can be used in as many situations as possible.

.NET also provides a lot of libraries, which can be referenced additionally, and it stands to reason that they are called class libraries or external libraries.

The Base Type Object Upcasting and Downcasting – Example

Let's take a closer look at the **Object class** using an example:

ObjectExample.cs

```
public class ObjectExample
{
  static void Main()
  {
    AfricanLion africanLion = new AfricanLion(true, 80);
    // Implicit casting
    object obj = africanLion;
  }
}
```

In this example, we cast an **AfricanLion** to **Object**. This operation is called **upcasting** and is permitted because **AfricanLion** is an indirect child of the **Object** class.

Now it is the time to mention that the keywords `string` and `object` are simply compiler tricks and are substituted with `System.String` and `System.Object` during compilation.

Let's continue with the example:

ObjectExample.cs

```
// …

AfricanLion africanLion = new AfricanLion(true, 80);
// Implicit casting
object obj = africanLion;

try
{
  // Explicit casting
  AfricanLion castedLion = (AfricanLion) obj;
}
catch (InvalidCastException ice)
{
  Console.WriteLine("obj cannot be downcasted to AfricanLion");
}
```

In this example, we **cast an `Object` to `AfricanLion`**. This operation is called **downcasting** and is permitted only if we indicate the type we want to cast to, because **`Object`** is a parent class of **`AfricanLion`** and it is not clear if the variable **`obj`** is of type **`AfricanLion`**. If it is not, an **`InvalidCastException`** will be thrown.

The Object.ToString() Method

One of the most commonly used methods, originating from the class **`Object`** is **`ToString()`**. It returns a **textual representation of an object**. Every object includes this method and therefore has a textual representation. This method is used when we print the object using **`Console.WriteLine(…)`**.

Object.ToString() – Example

Here is an example in which we call the **`ToString()`** method:

ToStringExample.cs

```
public class ToStringExample
{
  static void Main()
```

```
    {
        Console.WriteLine(new object());
        Console.WriteLine(new Felidae(true));
        Console.WriteLine(new Lion(true, 80));
    }
}
```

The result is:

```
System.Object
Chapter_20_OOP.Felidae
Chapter_20_OOP.Lion
```

In this case, the base class implementation is called, because **Lion doesn't override ToString()**. **Felidae** also doesn't override the method; therefore, we actually call the implementation **inherited from System.Object**. The result above contains the namespace of the object and the name of the class.

Overriding ToString() – Example

We will now demonstrate how useful overriding **ToString()** inherited from **System.Object** can be:

AfricanLion.cs

```
public class AfricanLion : Lion
{
    // …

    public override string ToString()
    {
        return string.Format(
            "(AfricanLion, male: {0}, weight: {1})",
            this.Male, this.Weight);
    }

    // …
}
```

In the source code above, we use the method **String.Format(…)**, in order to format the result appropriately. This is how we can then invoke the overridden method **ToString()**:

OverrideExample.cs

```
public class OverrideExample
```

```
{
   static void Main()
   {
      Console.WriteLine(new object());
      Console.WriteLine(new Felidae(true));
      Console.WriteLine(new Lion(true, 80));
      Console.WriteLine(new AfricanLion(true, 80));
   }
}
```

The result is:

```
System.Object
Chapter_20_OOP.Felidae
Chapter_20_OOP.Lion
(AfricanLion, male: True, weight: 80)
```

Notice that **ToString()** is invoked implicitly. When we pass an object to the **WriteLine()** method, that object provides its string representation using **ToString()** and only then it is printed to the output stream. That way, there's no need to explicitly get string representations of objects when printing them.

Virtual Methods: the "override" and "new" Keywords

We need to explicitly instruct the compiler that we want our method to **override** another. In order to do this, we use the **override keyword**. Notice what happens if we remove it:

```
public string ToString()
{
```

'AfricanLion.ToString()' hides inherited member 'object.ToString()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

```
}
```

Let's experiment and use the **keyword new** instead of **override**:

```
public class AfricanLion : Lion
{
   // …

   public new string ToString()
   {
      return string.Format(
```

```
        "(AfricanLion, male: {0}, weight: {1})",
        this.Male, this.Weight);
    }

    // …
}

public class OverrideExample
{
    static void Main()
    {
        AfricanLion africanLion = new AfricanLion(true, 80);
        string asAfricanLion = africanLion.ToString();
        string asObject = ((object)africanLion).ToString();
        Console.WriteLine(asAfricanLion);
        Console.WriteLine(asObject);
    }
}
```

This is the result:

```
(AfricanLion, male: True, weight: 80)
Chapter_20_OOP.AfricanLion
```

We notice that the implementation of **Object.ToString()** is invoked when we upcast **AfricanLion** to **object**. In other words, when we use the keyword **new**, we create a new method, which hides the old one. The old method can then only be called with an upcast.

What would happen, if we reverted to using the keyword **override** in the previous example? Take a look for yourself:

```
(AfricanLion, male: True, weight: 80)
(AfricanLion, male: True, weight: 80)
```

Surprising, isn't it? It turns out that when we override a method, we cannot access the old implementation even if we use upcasting. This is because there are no longer two **ToString()** methods, but rather only the one we overrode.

A method, which can be overridden, is called **virtual**. In .NET, methods are not **virtual** by default. If we want a method to be overridable, we can do so by including the keyword **virtual** in the declaration of the method.

The explicit instructions to the compiler that we want to override a method (by using **override**), is a protection against mistakes. If there's a typo in the method's name or the types of its parameters, the compiler will inform us

immediately of this mistake. It will know something is not right when it cannot find a method with the same signature in any of the base classes.

Virtual Methods are explained in details in the section about polymorphism.

Transitive Properties of Inheritance

In mathematics, **transitivity** indicates transferability of relationships. Let's take the indicator "larger than" (>) as an example. If A>B and B>C, we can **conclude** that A>C. This means that the relation "larger than" (>) is transitive, because we can unequivocally determine whether A is larger or smaller than C and vice versa.

If the class **Lion** inherits the class **Felidae** and the class **AfricanLion** inherits **Lion**, then this implies that **AfricanLion** inherits **Felidae**. Therefore, **AfricanLion** also has the property **Male**, which is defined in **Felidae**. This useful property allows a particular functionality to be defined in the most appropriate class.

Transitiveness – Example

Here is an example, which **demonstrates the transitive property of inheritance**:

TransitivenesExample.cs

```
public class TransitivityExample
{
    static void Main()
    {
        AfricanLion africanLion = new AfricanLion(true, 15);
        // Property defined in Felidae
        bool male = africanLion.Male;
        africanLion.Male = true;
    }
}
```

It is because of the transitive property of inheritance that we can be sure that all classes include the method **ToString()** and all other methods of **Object** regardless of which class they inherit.

Inheritance Hierarchy

If we try to describe all big cats, then, sooner or later, we will end up with a relatively large group of classes, which inherit one another. All these classes, combined with the base classes, form a **hierarchy** of big cat classes. The easiest way to describe such hierarchies is by using **class diagrams**. Let's take a look at what a "class-diagram" is.

Class Diagrams

A **Class Diagram** is one of several types of diagrams defined in UML. **UML (Unified Modeling Language)** is a notation for visualizing different processes and objects related to software development. We will talk about this further in the section on UML notation. Now let's discuss **class diagrams**, because they are used to describe visually class hierarchies, inheritance and the structure of the classes themselves.

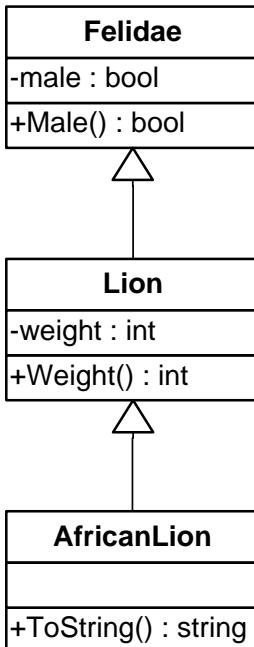
What is UML Class Diagram?

It is commonly accepted to draw class diagrams as **rectangles** with **name**, **attributes** (member variables) and **operations** (methods). The connections between them are denoted with various types of arrows.

Briefly, we will explain two pieces of UML terminology, so we can understand the examples more easily. The first one is **generalization**. Generalization is a term signifying the **inheritance of a class** or the **implementation of an interface** (we will explain interfaces shortly).

The other term is **association**. An association, would be, e.g. "The Lion has paws", where **Paw** is another class. Association is **has-a relationship**.

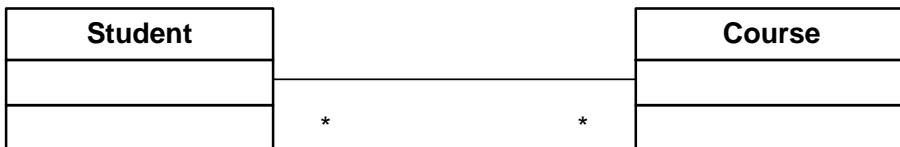
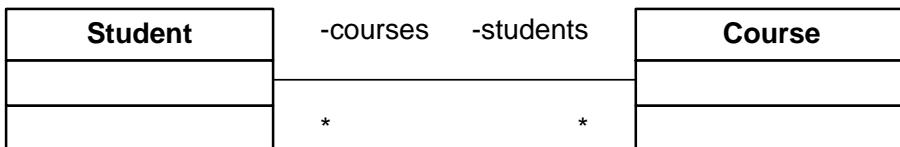
Generalization and association are the two main ways to reuse code.

A Class Based on a Class Diagram – Example

This is what a sample class diagram looks like:

Felidae
-male : bool
+Male() : bool

The class is represented as a **rectangle**, divided in 3 boxes one under another. The **name** of the class is at the top. Next, there are the **attributes** (UML term) of the class (in .NET they are called member variables and properties). At the very bottom are the **operations** (UML term) or methods (in .NET jargon). The plus/minus signs indicate whether an attribute / operation is visible (**+** means **public**) or not visible (**-** means **private**). **Protected** members are marked with **#**.

Class Diagram – Example of Generalization

Here is a class diagram that visually illustrates generalization (**Felidae** inherited by **Lion** inherited by **AfricanLion**):

In this example, the **arrows indicate generalization** (inheritance).

Associations

Associations denote **connections between classes**. They model mutual relations. They can define **multiplicity** (1 to 1, 1 to many, many to 1, 1 to 2, …, and many to many).

A **many-to-many** association is depicted in the following way:

A **many-to-many association by attribute** is depicted in the following way:

Student
-courses
-students
Course
*
*

In this case, there are connecting attributes, which indicate the variables holding the connection between classes.

A **one-to-many** association is depicted like this:

Student 1 — * Exam

A **one-to-one** association is depicted like this:

Country 1 — 1 Capital

From Diagrams to Classes

Class diagrams are most often used for creating classes. Diagrams facilitate and speed up the **design of classes in a software project**.

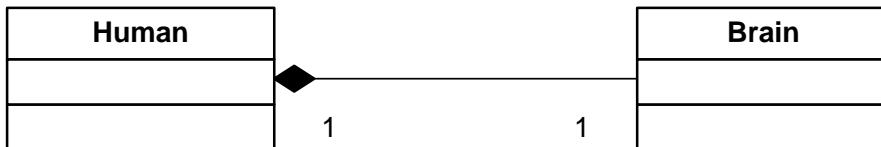
We can create classes directly following the diagram above. Here is the **`Capital`** class:

Capital.cs

```
public class Capital { }
```

And the **`Country`** class:

Country.cs

```
public class Country
{
   /// <summary>Country's capital - association</summary>
   private Capital capital;

   // …

   public Capital Capital
   {
      get { return capital; }
      set { this.capital = value; }
   }

   // …
}
```

Aggregation

Aggregation is a special type of association. It models the **relationship of kind "whole / part"**. We refer to the parent class as an **aggregate.** The aggregated classes are called **components**. There is an empty rhombus at one end of the aggregation:

Composition

A filled rhombus represents composition. Composition is an aggregation where the **components cannot exist without the aggregate**:

Abstraction

The next core principle of object-oriented programming we are about to examine is "abstraction". **Abstraction means working with something we know how to use without knowing how it works internally**. A good example is a television set. We don't need to know the inner workings of a TV, in order to use it. All we need is a remote control with a small set of buttons (the interface of the remote) and we will be able to watch TV.

The same goes for objects in OOP. If we have an object **`Laptop`** and it needs a processor, we use the object **`Processor`**. We do not know (or rather it is of no concern to us) how it calculates. In order to use it, it's sufficient to call the method **`Calculate()`** with appropriate parameters.

Abstraction is something we do every day. This is an action, which obscures all details of a certain object that do not concern us and only uses the details, which are relevant to the problem we are solving. For example, in hardware configurations, there is an **abstraction called "data storage device"** which can be a **hard disk**, **USB memory stick** or **CD-ROM drive**. Each of these works in a different way internally but, from the point of view of the operating system and its applications, it is used in the same way – it **stores files and folders**. In Windows we have Windows Explorer and it can work with all devices in the same way, regardless of whether a device is a hard drive or a USB stick. It works with the **abstraction** "storage device" and is not involved with how data is read or written. The drivers of the particular device take care of that. They are implementations of the interface "data storage device".

Abstraction is one of the **most important concepts** in programming and OOP. It allows us to write **code, which works with abstract data structures** (like dictionaries, lists, arrays and others). We can work with an abstract data type by using its interface without concerning ourselves with its implementation. For instance, we can save to a file all elements from a list without bothering if it is implemented with an array, a linked list, etc. The code remains unchanged, when we work with other data types. We can even write new data types (we will discuss this later) and make them work with our program without changing it.

Abstraction allows us to do something very important – **define an interface for our applications**, i.e. to define a**ll tasks the program is capable to execute** and their respective input and output data. That way we can make a couple of small programs, each handling a smaller task. When we combine this with the ability to work with **abstract data**, we achieve great flexibility in integrating these small programs and much more opportunities for code reuse. These small subprograms are referred to as **components**. This approach for writing programs is widely adopted since it allows us to reuse not only objects, but entire subprograms as well.

Abstraction – Abstract Data Example

Here is an example, where we define a specific data type "African lion", but use it later on in an abstract manner through the "Felidae" abstraction. This **abstraction** does not concern itself with the details of all types of lions.

AbstractionExample.cs

```
public class AbstractionExample
{
   static void Main()
   {
      Lion lion = new Lion(true, 150);
      Felidae bigCat1 = lion;

      AfricanLion africanLion = new AfricanLion(true, 80);
      Felidae bigCat2 = africanLion;
   }
}
```

Interfaces

In the C# language the **interface** is a definition of a **role** (a group of abstract actions). It defines what sort of behavior a certain object must exhibit, without specifying how this behavior should be implemented. Interfaces are also known as **contracts** or **specifications** of behavior.

An object can have **multiple roles** (or **implement multiple interfaces** / contracts) and its users can utilize it from different points of view.

For example, an object of type **Person** can have the roles of **Soldier** (with behavior "shoot your enemy"), **Husband** (with behavior "love your wife") and **Taxpayer** (with behavior "pay your taxes"). However, every person implements its behavior in a different way; **John** pays his taxes on time, **George** pays them overdue and **Peter** doesn't pay them at all.

Some may ask why the base class of all objects (the class **Object**) is not an interface. The reason is because in such case, every class would have to implement a small, but very important group of methods and this would take an unnecessary amount of time. It turns out that not all classes need a specific implementation of **Object.GetHashCode()**, **Object.Equals(…)** and **Object.ToString()**, i.e. the default implementation suffices in most cases. It's not necessary to override any of the methods in the **Object** class, but if the situation calls for it we can. **Overriding methods** is explained in the virtual methods section.

Interfaces – Key Concepts

An **interface can only declare methods and constants**.

A **method signature** is the combination of a method's **name** and a description of its **parameters** (type and order). In a class / interface all methods have to have different signatures and should not be identical with signatures of inherited methods.

A **method declaration** is the combination of a method's **return type and its signature**. The return type only specifies what the method returns.

A method is identified by its signature. The return type is not a part of it. If two methods' only difference is the return type (as in the case when a class inherits another), then it cannot be unequivocally decided which method must be executed.

A **class / method implementation** is the source code of a class / method. Usually it is between curly brackets: "**{**" and "**}**". Regarding methods, this is also referred to as the **method body**.

Interfaces – Example

An interface in .NET is defined with the **keyword interface**. An interface can contain only method declarations and constants. Here is an example of an interface:

Reproducible.cs

```
public interface Reproducible<T> where T : Felidae
{
```

```
    T[] Reproduce(T mate);
}
```

We explained the generics in the "Defining Classes" chapter (section "Generics"). The interface we wrote has a method of type **T** (**T** must inherit **Felidae**) which returns an array of **T**.

And this is how the class **Lion**, which **implements the interface Reproducible** looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    // …

    Lion[] Reproducible<Lion>.Reproduce(Lion mate)
    {
        return new Lion[]{new Lion(true, 12), new Lion(false, 10)};
    }
}
```

The name of the interface is coded in the declaration of the class (on the first row) and specifies the generic class.

We can indicate which method from a specific interface we implement by typing its name explicitly:

```
Lion[] Reproducible<Lion>.Reproduce(Lion mate)
```

In an interface, methods are only declared; the implementation is coded in the class implementing the interface, i.e. – **Lion**.

The class that implements a certain interface must **implement all methods in it**. The only exception is when the class is **abstract**. Then it can implement none, some or all of the methods. All remaining methods have to be implemented in some of the inheriting classes.

Abstraction and Interfaces

The best way to achieve abstraction is by **working though interfaces**. A component works with interfaces which another implements. That way, a change in the second component will not affect the first one as long as the new component implements the old interface. The interface is also called a **contract**. Every component upholds a certain contract (the signature of certain methods). That way, two components upholding a contract can communicate with each other without knowing how their counterpart works.

Some important interfaces from the Common Type System (CTS) are the **list** and **collection** interfaces: **System.Collections.Generic.IList<T>** and **System.Collections.Generic.ICollection<T>**. All of the standard .NET collection classes implement these interfaces and the various components pass different implementations (arrays, linked lists, hash tables, etc.) to one another using a common interface.

Collections are an excellent example of an object-oriented library with classes and interfaces that actively use all core principles of OOP: abstraction, inheritance, encapsulation and polymorphism.

When Should We Use Abstraction and Interfaces?

The answer to this question is: **always when we want to achieve abstraction of data or actions**, whose implementation can change later on. Code, which communicates with another piece of code through interfaces, is much more resilient to changes than code written using specific classes. **Working through interfaces is common and a highly recommended practice** – one of the basic rules for writing high-quality code.

When Should We Write Interfaces?

It is always a good idea to use interfaces **when functionality is exposed to another component**. In the interface we include only the functionality (in the form of a declaration) that others need to see.

Internally, a program / component can use interfaces for **defining roles**. That way, an object can be used by different classes through different roles.

Encapsulation

Encapsulation is one of the main concepts in OOP. It is also called "**information hiding**". An object has to provide its users only with the essential information for manipulation, without the internal details. A **Secretary** using a **Laptop** only knows about its screen, keyboard and mouse. Everything else is hidden internally under the cover. She **does not know about the inner workings** of **Laptop**, because she doesn't need to, and if she does, she might make a mess. Therefore parts of the properties and methods remain hidden to her.

The person writing the class has to decide what should be **hidden** and what not. When we program, we must define as **private** every method or field which other classes should not be able to access.

Encapsulation – Examples

The example below shows how to **hide methods** that the class' user doesn't have to be familiar with and are only used internally by the author of the class. First, we define an **abstract class Felidae**, which defines the public operations of cats (regardless of the cat's type):

Felidae.cs

```
public class Felidae
{
    public virtual void Walk()
    {
        // …
    }

    // …
}
```

This is how the class **Lion** looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    // …

    private Paw frontLeft;
    private Paw frontRight;
    private Paw bottomLeft;
    private Paw bottomRight;

    private void MovePaw(Paw paw) {
        // …
    }

    public override void Walk()
    {
        this.MovePaw(frontLeft);
        this.MovePaw(frontRight);
        this.MovePaw(bottomLeft);
        this.MovePaw(bottomRight);
    }

    // …
}
```

The public method **Walk()** calls some other **private** method 4 times. That way the base class is short – it consists of a single method. The implementation, however, calls another of its methods, which is hidden from the users of the class. That way, **Lion doesn't publicly disclose information about its inner workings** (it encapsulates certain behavior).

At a later stage, this makes it possible to change its implementation without any of the other classes finding out and requiring changes.

Polymorphism

The next fundamental principle of Object-Oriented Programming is "Polymorphism". **Polymorphism allows treating objects of a derived class as objects of its base class**. For example, big cats (base class) catch their prey (a method) in different ways. A Lion (derived class) sneaks on it, while a Cheetah (another derived class) simply outruns it.

Polymorphism allows us to treat a cat of random size just like a big cat and command it "catch your prey", regardless of its exact size.

Polymorphism can bear strong resemblance to abstraction, but it is mostly related to **overriding methods in derived classes**, in order **to change their original behavior** inherited from the base class. Abstraction is associated with creating an interface of a component or functionality (defining a role). We are going to explain method overriding shortly.

Abstract Classes

What happens if we want to specify that the class **Felidae** is **incomplete** and only its successors can have instances? This is accomplished by putting **the keyword abstract** before the name of the class and indicates that the class is **not ready to be instantiated**. We refer to such classes as **abstract classes**. And how do we indicate which exact part of the class is incomplete? Once again, this is accomplished by putting the keyword **abstract** before the name of the method to be implemented. This method is called an **abstract method** and cannot have an implementation, but a declaration only.

Each class with **at least one abstract method** must be abstract. Makes sense, right? However, the opposite is not true. It is possible to define a class as an abstract one, even when there are no abstract methods in it.

Abstract classes are something **in the middle between classes and interfaces**. They can define **ordinary methods** and **abstract methods**. Ordinary methods have an implementation, whereas abstract methods are **empty** (without an implementation) and remain to be implemented later by the derived classes.

Abstract Class – Examples

Let's take a look at an **example of an abstract class**:

Felidae.cs

```
/// <summary>Latin for "cats"</summary>
public abstract class Felidae
{
```

```
    // …

    protected void Hide()
    {
        // …
    }

    protected void Run()
    {
        // …
    }

    public abstract bool CatchPrey(object prey);
}
```

Notice how in the example above the ordinary methods **Hide()** and **Run()** have a body, while the abstract method **CatchPrey()** does not. Notice that the methods are declared as **protected**.

Here is how the **implementation** of the above abstraction looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
    protected void Ambush()
    {
        // …
    }

    public override bool CatchPrey(object prey)
    {
        base.Hide();
        this.Ambush();
        base.Run();
        // …
        return false;
    }
}
```

Here is one more example of **abstract behavior**, implemented with an abstract class and a polymorphic call to an abstract method. In this example we define abstract method and we override it later in a descendant class. Let's see the code and discuss it later.

Firstly, we define the abstract class **Animal**:

Animal.cs

```
public abstract class Animal
{
  public void PrintInformation()
  {
    Console.WriteLine("I am a {0}.", this.GetType().Name);
    Console.WriteLine(GetTypicalSound());
  }

  protected abstract String GetTypicalSound();
}
```

We also define the class **Cat**, which **inherits the abstract class Animal** and defines an implementation of the abstract method **GetTypicalSound()**:

Cat.cs

```
public class Cat : Animal
{
  protected override String GetTypicalSound()
  {
    return "Meoooow!";
  }
}
```

If we execute the following program:

```
public class AbstractClassExample
{
  static void Main()
  {
    Animal cat = new Cat();
    cat.PrintInformation();
  }
}
```

we are going to get the following result:

```
I am a Cat.
Meoooow!
```

In the example, the **PrintInformation()** method from the abstract class does its work by relying on the result from a call to the **abstract method GetTypicalSound()** which is expected to be implemented in different ways by

the kinds of animals (the various successors of the class **Animal**). Different animals make distinct sounds, but the functionality for printing information about animals is common to all animals, and that's why it is exported to the base class.

Purely Abstract Classes

Abstract classes, as well as interfaces, **cannot be instantiated**. If we try to create an instance of an abstract class, we are going to get an error during compilation.

Sometimes a class can be declared abstract, even if it has no abstract methods, in order to simply prohibit using it directly without creating an instance of a successor.

A **pure abstract class** is an abstract class, which has no implemented methods and no member variables. It is **very similar to an interface**. The fundamental difference is that a class can implement many interfaces and inherit only one class (even if that class is abstract).

Initially, interfaces were not necessary in the presence of "multiple inheritance". They had to be conceived as a means to supersede it in specifying the numerous roles of an object.

Virtual Methods

A method, which can be overridden in a derived class, is called a **virtual method**. Methods in .NET by default aren't virtual. If we want to make a method virtual, we mark it with **the keyword virtual**. Then the derived class can declare and define a method with the same signature.

Virtual methods are important for **method overriding**, which lies at the heart of polymorphism.

Virtual Methods – Example

We have a class inheriting another and the two classes share a common method. Both versions of the method write on the console. Here is how the **Lion** class looks like:

Lion.cs

```
public class Lion : Felidae, Reproducible<Lion>
{
  public override void CatchPrey(object prey)
  {
    Console.WriteLine("Lion.CatchPrey");
  }
}
```

Here is how the **AfricanLion** class looks like:

AfricanLion.cs

```
public class AfricanLion : Lion
{
    public override void CatchPrey(object prey)
    {
        Console.WriteLine("AfricanLion.CatchPrey");
    }
}
```

We make three attempts to create instances and call the method **CatchPrey**.

VirtualMethodsExample.cs

```
public class VirtualMethodsExample
{
    static void Main()
    {
        Lion lion = new Lion(true, 80);
        lion.CatchPrey(null);
        // Will print "Lion.CatchPrey"

        AfricanLion lion = new AfricanLion(true, 120);
        lion.CatchPrey(null);
        // Will print "AfricanLion.CatchPrey"

        Lion lion = new AfricanLion(false, 60);
        lion.CatchPrey(null);
        // Will print "AfricanLion.CatchPrey", because
        // the variable lion has a value of type AfricanLion
    }
}
```

In the last attempt, you can clearly see how, in fact, **the overwritten method is called** and not the base method. This happens, because it is validated what the actual class behind the variable is and whether it implements (overwrites) that method. **Rewriting of methods** is also called **overriding of virtual methods**.

Virtual methods as well as abstract methods can be overridden. Abstract methods are actually virtual methods without a specific implementation. All methods defined in an interface are abstract and therefore virtual, although this is not explicitly defined.

Virtual Methods and Methods Hiding

In the example above, the implementation of the base class is hidden and omitted. Here is how we can also use it as part of the new implementation (in case we want to complement the old implementation rather than override it).

Here is how the **AfricanLion** class looks like:

AfricanLion.cs

```
public class AfricanLion : Lion
{
   public override void CatchPrey(object prey)
   {
      Console.WriteLine("AfricanLion.CatchPrey");
      Console.WriteLine("calling base.CatchPrey");
      Console.Write("\t");
      base.CatchPrey(prey);
      Console.WriteLine("...end of call.");
   }
}
```

In this example, three lines will be written on the console when **AfricanLion.CatchPrey(…)** is called:

```
AfricanLion.CatchPrey
calling base.CatchPrey
     Lion.CatchPrey
...end of call.
```

The Difference between Virtual and Non-Virtual Methods

Some may ask what the difference between the virtual and non-virtual methods is.

Virtual methods are used when we expect from derived classes to change / complement / **alter some of the inherited functionality**. For example, the method **Object.ToString()** allows derived classes to change / replace its implementation in any way they want. Then, even if we work with an object not directly, but rather by upcasting it to **Object**, we use the overwritten implementation of the virtual methods.

Virtual methods are a key characteristic of objects when we talk about abstraction and working with abstract types.

Sealing of methods is done when we rely on a piece of functionality and we don’t want it to be altered. We already know that methods are **sealed** by default. But if we want a base class’ virtual method to become sealed in a derived class, we use **override sealed**.

The **string class has no virtual methods**. In fact, inheriting **string** is entirely **forbidden for inheritance** through the keyword **sealed** in its declaration. Here are parts of the declarations of **string** and **object** classes (the ellipses in square brackets indicate omitted, irrelevant code):

```
namespace System
{
  […] public class Object
  {
    […] public Object();
    […] public virtual bool Equals(object obj);
    […] public static bool Equals(object objA, object objB);
    […] public virtual int GetHashCode();
    […] public Type GetType();
    […] protected object MemberwiseClone();
    […] public virtual string ToString();
  }

  […] public sealed class String : […]
  {
    […] public String(char* value);
    […] public int IndexOf(string value);
    […] public string Normalize();
    […] public string[] Split(params char[] separator);
    […] public string Substring(int startIndex);
    […] public string ToLower(CultureInfo culture);
    […]
  }
}
```

When Should We Use Polymorphism?

The answer to this question is simple: whenever we want **to enable changing a method's implementation in a derived class**. It's a good rule to work with the most basic class possible or directly with an interface. That way, changes in used classes reflect to a much lesser extent on classes written by us. The less a program knows about its surrounding classes, the fewer changes (if any) it would have to undergo.

Cohesion and Coupling

The terms **cohesion and coupling** are inseparable from OOP. They complement and explain further some of the principles we have described so far. Let's get familiar with them.

Cohesion

The concept of **cohesion** shows to what degree a program's or a component's various tasks and responsibilities are **related to one another**, i.e. how much a program is focused on solving a single problem. Cohesion is divided into **strong cohesion** and **weak cohesion**.

Strong Cohesion

Strong cohesion indicates that the responsibilities and tasks of a piece of code (a method, class, component or a program) are **related to one another** and intended to **solve a common problem**. This is something we must always aim for. Strong cohesion is a typical characteristic of high-quality software.

Strong Cohesion in a Class

Strong cohesion in a class indicates that the class **defines only one entity**. We mentioned earlier that an entity can have many roles (Peter is a soldier, husband and a taxpayer). Each of these roles is defined in the same class. Strong cohesion indicates that the class solves only one task, one problem, and not many at the same time.

A class, which does **many things at the same time**, is difficult to understand and maintain. Consider a class, which implements a hash table, provides functions for printing, sending an e-mail and working with trigonometric functions all at once. How do we name such a class? If we find it difficult to answer this question, this means that we have failed to achieve **strong cohesion** and have to separate the class into several smaller classes, each solving a single task.

Strong Cohesion in a Class – Example

As an example of strong cohesion we can point out the **`System.Math`** class. It performs **a single task**: it provides mathematical calculations and constants:

- **`Sin()`**, **`Cos()`**, **`Asin()`**
- **`Sqrt()`**, **`Pow()`**, **`Exp()`**
- **`Math.PI`**, **`Math.E`**

Strong Cohesion in a Method

A method is well written when it performs only one task and performs it well. A method, which does a lot of work related to different things, has **bad cohesion**. It has to be **broken down into simpler methods**, each solving only one task. Once again, the question is posed what name should we give to a method, which finds prime numbers, draws 3D graphics on the screen, communicates with the network and prints records extracted from a data base? Such a method has **bad cohesion** and has to be logically separated into several methods.

Weak Cohesion

Weak cohesion is observed along with **methods, which perform several unrelated tasks**. Such methods take several different groups of parameters, in order to perform different tasks. Sometimes, this requires logically unrelated data to be unified for the sake of such methods. Weak cohesion is harmful and must be avoided!

Weak Cohesion – Example

Here is a sample class with weak cohesion:

```
public class Magic
{
  public void PrintDocument(Document d) { … }
  public void SendEmail(string recipient,
    string subject, string text) { … }
  public void CalculateDistanceBetweenPoints(
    int x1, int y1, int x2, int y2) { … }
}
```

Best Practices with Cohesion

Strong cohesion is quite logically the "good" way of writing code. The concept is associated with simpler and clearer source code – code that is easier to maintain and reuse (because of the fewer tasks it has to perform).

Contrarily, with **weak cohesion** each change is a ticking time bomb, because it could affect other functionality. Sometimes a logical task is spread out to several different modules and thus changing it is more labor intensive. Code reuse is also difficult, because a component does several unrelated tasks and to reuse it the exact same conditions must be met which is hard to achieve.

Coupling

Coupling mostly describes the extent to which components / classes **depend on one another**. It is broken down into **loose coupling** and **tight coupling**. Loose coupling usually correlates with strong cohesion and vice versa.

Loose Coupling

Loose coupling is defined by a piece of code's (program / class / component) communication with other code through **clearly defined interfaces** (contracts). A change in the implementation of a loosely coupled component doesn't reflect on the others it communicates with. When you write source code, you **must not rely on inner characteristics** of components (specific behavior that is not described by interfaces).

The contract has to be maximally simplified and define only the required behavior for this component's work by hiding all unnecessary details.

Loose coupling is a code characteristic you should aim for. It is one of the characteristics of high-quality programming code.

Loose Coupling – Example

Here is an **example of loose coupling** between classes and methods:

```
class Report
{
  public bool LoadFromFile(string fileName) { … }
  public bool SaveToFile(string fileName) { … }
}

class Printer
{
  public static int Print(Report report) { … }
}

class Example
{
  static void Main()
  {
    Report myReport = new Report();
    myReport.LoadFromFile("DailyReport.xml");
    Printer.Print(myReport);
  }
}
```

In this example, **none of the methods depend on the others**. The methods rely only on some of the parameters, which are passed to them. Should we need one of the methods in a next project, we could easily take it out and reuse it.

Tight Coupling

We achieve tight coupling when there are many input parameters and output parameters; when we use undocumented (in the contract) characteristics of another component (for example, a dependency on static fields in another class); and when we use many of the so called control parameters that indicate behavior with actual data. **Tight coupling** between two or more methods, classes or components means that **they cannot work independently of one another** and that a change in one of them will also affect the rest. This leads to difficult to read code and big problems with its maintenance.

Tight Coupling – Example

Here is an **example of tight coupling** between classes and methods:

```
class MathParams
{
  public static double operand;
  public static double result;
}

class MathUtil
{
  public static void Sqrt()
  {
    MathParams.result = CalcSqrt(MathParams.operand);
  }
}

class SpaceShuttle
{
  static void Main()
  {
    MathParams.operand = 64;
    MathUtil.Sqrt();
    Console.WriteLine(MathParams.result);
  }
}
```

Such code is **difficult to understand and maintain**, and the likelihood of mistakes when using it is great. Think about what happens if another method, which calls **Sqrt()**, passes its parameters through the same static variables **operand** and **result**.

If we have to use the same functionality for deriving square root in a subsequent project, we will not be able to simply copy the method **Sqrt()**, but rather we will have to copy the classes **MathParams** and **MathUtil** together with all of their methods. This makes the code difficult to reuse.

In fact, the above code is an **example of bad code** according to all rules of Procedural and Object-Oriented Programming and if you think twice, you will certainly identify at least several more disregarded recommendations from those we have given you so far.

Best Practices with Coupling

The most common and advisable way of invoking a well written module's functionality is through interfaces. That way, the functionality can be substituted without clients of the code requiring changes. The jargon expression for this is "**programming against interfaces**".

Most commonly, an interface describes a "**contract**" observed by this module. It is good practice not to rely on anything else other than what's described by

this contract. The use of inner classes, which are not part of the public interface of a module, is not recommended because their implementation can be substituted without substituting the contract (we already discussed this in the section "Abstraction").

It is good practice that the **methods are made flexible** and ready to work with all components, which observe their interfaces, and not only with definitive ones (i.e. to have implicit requirements). The latter would mean that these methods expect something specific from the components they can work with. It is also good practice that **all dependencies are clearly described and visible**. Otherwise, the maintenance of such code becomes difficult (it is riddled with stumbling-blocks).

A good example of strong cohesion and loose coupling we can find in the classes from the standard namespaces **`System.Collections`** and **`System.Collections.Generic`**. These .NET classes for working with collections have **strong cohesion**. Each solves a single problem and allows easy reuse. These classes have another characteristic of high-quality programming code: **loose coupling**. The classes, implementing the collections, are not related to one another. Each works **through a strictly defined interface** and **does not give away details of its implementation**. All methods and fields not from the interface are hidden, in order to reduce the possibility of coupling with them. Methods in the collection classes do not depend on static variables and do not rely on any input data except for their inner state and passed parameters. This is good practice every programmer sooner or later attains with gained experience.

Spaghetti Code

Spaghetti code is **unstructured code with unclear logic**; it is **difficult to read, understand and maintain**; it violates and mixes up consistency; it has **weak cohesion and tight coupling**. Such code is associated with spaghetti, because it is just as tangled and twisted. When you pull out a strand of spaghetti (i.e. a class or method), the whole dish of spaghetti can turn out tangled in it (i.e. changes in one method or class lead to dozens of other changes because of the strong dependence between them). It is almost impossible to reuse spaghetti code, since there is no way to separate that part of the code, which is practically applicable.

Spaghetti code is achieved when you have written code, supplement it and have to readapt it again and again every time the requirements change. Time passes by until a moment comes when it has to be rewritten from scratch.

Cohesion and Coupling in Engineering Disciplines

If you think that the principles of strong cohesion and loose coupling apply only to programming, you are deeply mistaken. These are **fundamental engineering principles** you will come across in construction, machine building, electronics and thousands of other fields.

Let's take, for instance, a **hard disk drive (HDD)**:

It **solves only one task** doesn't it? The hard disk solves the task of storing data. It does not cool down the computer, does not make sounds, has no computing power and is not used as a keyboard. It is connected to the computer with two cables only, i.e. it has a **simple interface** for access and is not bound to other peripherals. The hard disk **works separately** and other devices aren't concerned about how it works exactly. The CPU commands it to "read" and it reads, then it commands it to "write" and it writes. How exactly it does this remains **hidden inside** it. Different models can work in different ways, but that is their own concern. You can see that the CPU has strong cohesion, loose coupling, good abstraction and good encapsulation. This is how you should implement your classes – they must **do only one thing**, **do it well**, **bind them minimally to other classes** (or not link them at all whenever that's possible), have a clear interface and good abstraction and to hide the details of their internal workings.

Here is another example: imagine what would happen, if the processor, the hard disk, the CD-ROM drive and the keyboard were soldered to the motherboard of the computer. It would mean that if any part of the keyboard were broken, you would have to throw away the whole computer. You can see how hardware cannot work well with tight coupling and weak cohesion. The same applies to software.

Object-Oriented Modeling (OOM)

Suppose we have a problem or task to solve. The problem usually comes from the real word. It exists in a reality we are going to call its surrounding environment.

Object-oriented modeling (OOM) is a process associated with OOP where all objects related to the problem we are solving are brought out (a model is created). Only the classes' characteristics, which are important for solving this particular problem, are elicited. The rest are ignored. That way, we create a new reality, a **simplified version of the original** one (its model), such that it allows us to solve the problem or task.

For example, if we model a **ticketing system**, the **important characteristics** of a passenger could be their name, their age, whether they use a discount and whether they are male or female (if we sell sleeping

berths). A passenger has **many other not important characteristics we aren't concerned about**, such as the color of their eyes, what shoe size they wear, what books they like or what beer they drink.

By modeling, **a simplified model of reality is created** in order to solve a specific task. In object-oriented modeling, the model is created by means of OOP: via classes, class attributes, class methods, objects, relations between classes, etc. Let's scrutinize this process.

Steps in Object-Oriented Modeling

Object-oriented modeling is usually performed in these steps:

- Identification of classes.
- Identification of class attributes.
- Identification of operations on classes.
- Identification of relations between classes.

We will consider a short **example** through which we will demonstrate how to apply these steps.

Identification of Classes

Suppose we have the following excerpt from a system's **specification**:

> The user must be able to describe each product by its characteristics, including name and product number. If the barcode doesn't match the product, an error must be generated on the error screen. There has to be a daily report for all transactions specified in section 9.3.

Here is how we identify key concepts:

> The **user** must be able to describe each **product** by its **characteristics**, including **name** and **product number**. If the **barcode** doesn't match the product, an **error** must be generated on the **error screen**. There has to be a **daily report** for all **transactions** specified in section 9.3.

We have just **identified the classes** we will need. The names of the classes are **the nouns in the text**, usually common nouns in singular like **`Student`**, **`Message`**, **`Lion`**. Avoid names that don't come from the text, such as: **`StrangeClass`**, **`AddressTheStudentHas`**.

Sometimes it's difficult to determine whether some subject or phenomena from the real world has to be a class. For example, the **address** can be defined as a **class `Address` or a string**. The better we explore the problem, the easier it will be to decide which entities must be represented as classes. When a class becomes large and complicated it has to be broken down into several smaller classes.

Identification of Class Attributes

Classes have **attributes (characteristics)**, for example the class **Student** has a name, institution and a list of courses. Not all characteristics are important for a software system. For example, as far as the class **Student** is concerned eye color is a non-essential characteristic. **Only essential characteristics have to be modeled**.

Identification of Operations on Classes

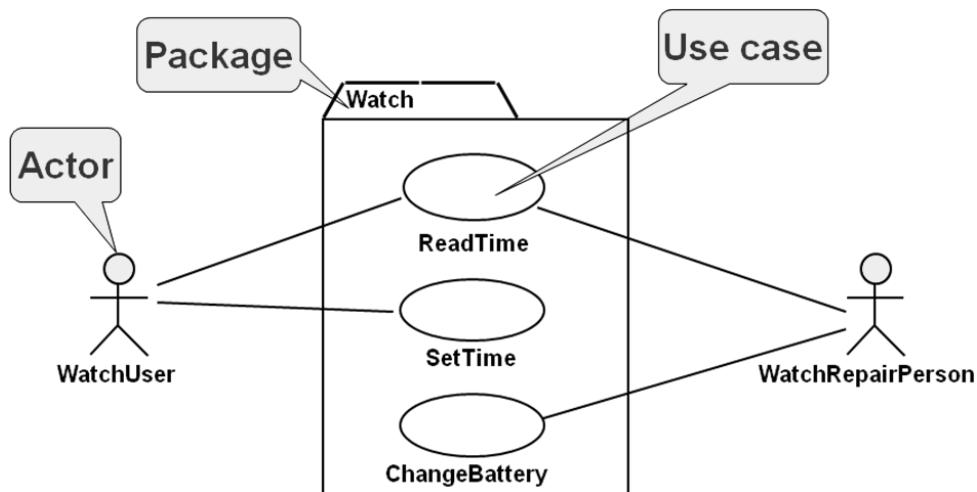
Each class must have **clearly defined responsibilities** – what objects or processes from the real world it identifies and what tasks it performs. Each action in the program is performed by one or several methods in some class. The actions are modeled as operations (methods).

A combination of **verb + noun is used for the name of a method**, e.g. **PrintReport()**, **ConnectToDatabase()**. We cannot define all methods of a given class immediately. Firstly, we define the most important methods – those that implement the basic responsibilities of the class. Over time additional methods appear.

Identification of Relationships between Classes

If a student is from a faculty and this is important for the task we are solving, then student and faculty are related, i.e. the **Faculty** class has a list of **Students**. These relations are called **associations** (remember the "Class Diagrams" section).

UML Notation

UML (Unified Modeling Language) was mentioned in the section about inheritance where we discussed class diagrams. The UML notation defines several additional types of diagrams. Let's check out some of them briefly.

Use Case Diagrams

They are used when we elicit the requirements for the description of possible actions. **Actors** represent roles (types of users).

Use cases describe interaction between the actors and the system. The use case model is a group of use cases – it provides a complete description of a system's functionality.

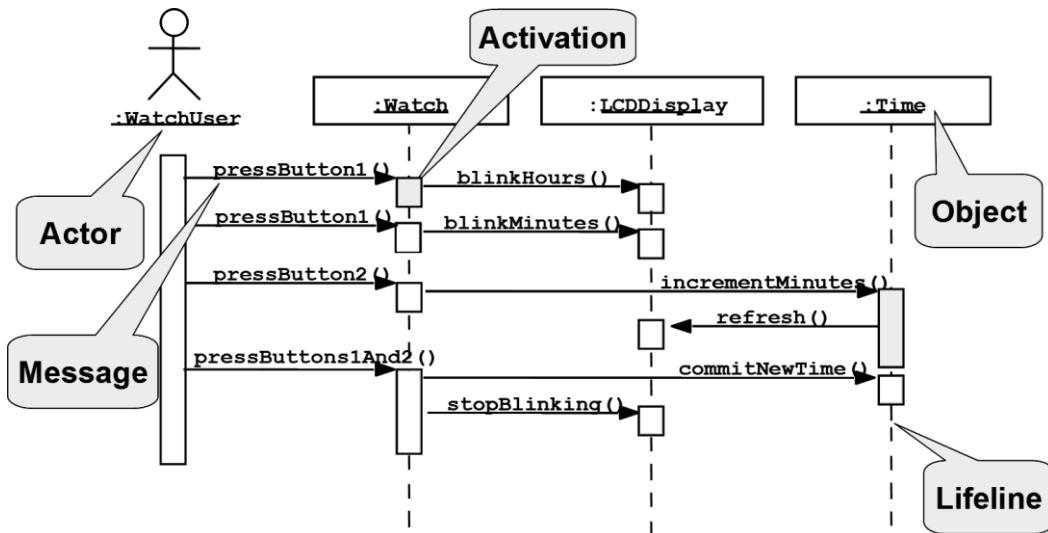
Use Case Diagrams – Example

Here is how a **use case diagram** looks like:

The **actor** (the “dwarf” in the diagram) is someone who interacts with the system (a user, external system or, for instance, an external environment). The **actor has a unique name** and, possibly, a **description**. I our case actors are the **`WatchUser`** and the **`WatchRepairPerson`**.

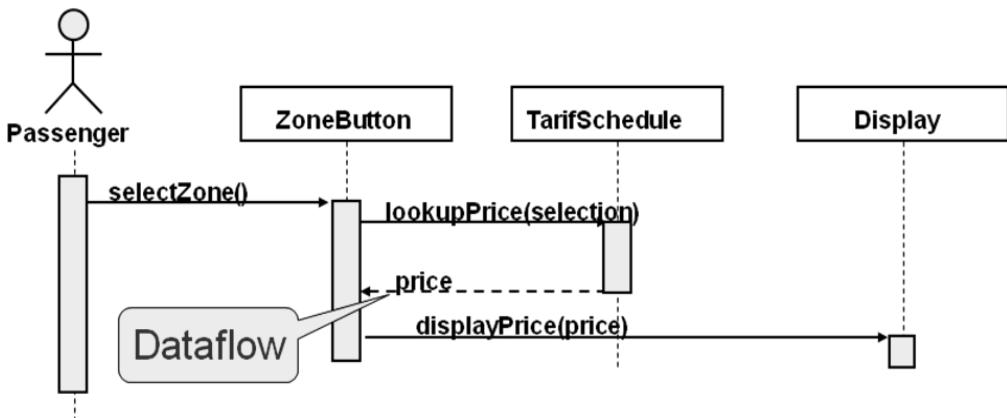
A **use case** (the “egg” in the diagram) describes a **single functionality of the system**, a single **action** that can be performed by some actor. It has a **unique name** and is **related to actors**. It can have input and output conditions. Most frequently, it contains a flow of operations (a process). It can also have other requirements. We have three use cases in the diagram above: **`ReadTime`**, **`SetTime`** and **`ChangeBattery`**.

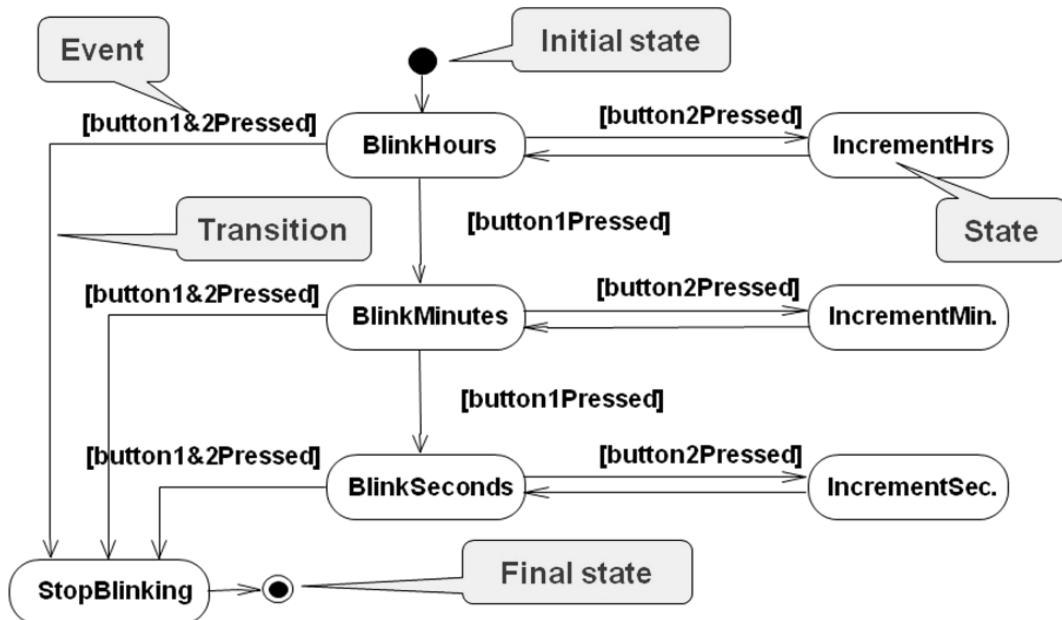
A **package** holds several logically related use cases.

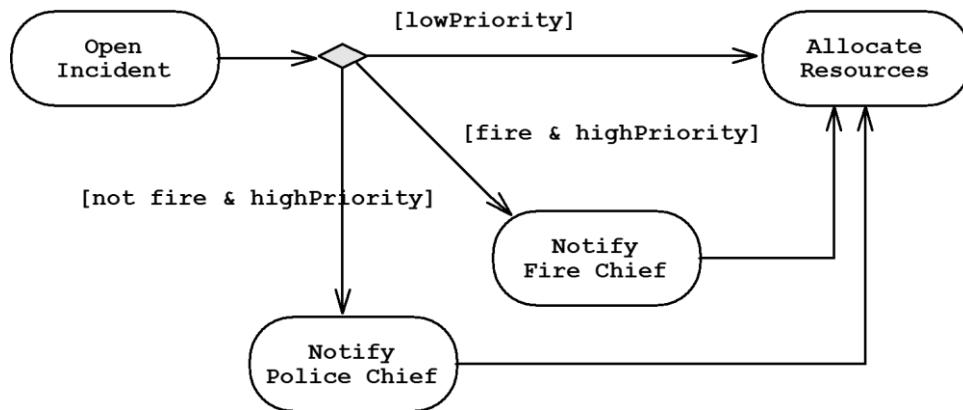
Lines connect actors to the use cases they perform. An actor can perform or be involved in one or several use cases.

Sequence Diagrams

Sequence diagrams are used when modeling the requirements of **process specification** and describing use case scenarios more extensively. They allow describing additional participants in the processes and the sequence of the actions over the time. They are used in designing the descriptions of system interfaces.

Sequence diagrams describe **what happens over the time**, the interactions over the time, the **dynamic view** over the system, a **sequence of steps**, just like an algorithm.

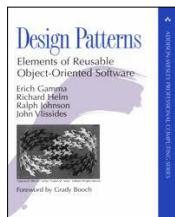
Sequence Diagrams – Example

Here is how a sequence diagram looks like:

Classes are depicted with columns (lifelines). **Messages (actions)** are depicted with arrows and text above the arrows. **Participants** are depicted with wide rectangles. **States** are depicted with dashed lines. The period of activity (**activatio**n) of certain class during the time is depicted as narrow rectangles.

Messages – Example

The direction of the arrow designates the **sender** and the **recipient** of a **message** (a method call in OOP). Horizontal dashed lines depict **data flow**:

Statechart Diagrams

Statechart diagrams describe the possible **states** of certain process and the possible **transitions** between them along with the conditions for the transitions. They represent **finite-state automata** (**state machines**). Below we have an **example of statechart diagram** that illustrates the states and transitions of typical process of changing the current time of a wall clock which has two buttons and a screen:

Activity Diagrams

Activity diagrams are a **special type of statechart diagrams** where **conditions are actions**. They show the flow of actions in a system:

Design Patterns

Few years after the onset of the object-oriented paradigm it was found that there are many situations, which occur frequently during software development, such as a class, which must have only one instance within the entire application.

Design patterns appeared as proven and highly-efficient **solutions to the most common problems of object-oriented modeling**. Design patterns are systematically described in the eponymous book by Erich Gamma & Co.

"**Design Patterns: Elements of Reusable Object-Oriented Software**" (ISBN 0-201-63361-2). The patterns in this book are called "**the GoF patterns**" or "classical design patterns".

This is one of the few books in the field of computer science, which remain current 15 years after publishing. Design patterns complement the basic principles of OOP with **well-known solutions of well-known problems**. A good place to start studying the design patterns is their Wikipedia article: en.wikipedia.org/wiki/Design_pattern_(computer_science). You may also check the "Data & Object Factory" **patterns catalog** http://www.dofactory.com/Patterns/Patterns.aspx, where the authors provide C# implementation of the classical GoF patterns.

The Singleton Design Pattern

This is the most popular and most frequently used design pattern. It allows a **class to have only one instance** and defines where it has to be taken from. Typical examples are classes, which define references to singular entities (a virtual machine, operating system, window manager in a graphical application or a file system) as well as classes of the next pattern (factory).

The Singleton Design Pattern – Example

Here is a sample implementation of the **singleton** design pattern:

Singleton.cs

```
public class Singleton
{
   // The single instance
   private static Singleton instance;

   // Initialize the single instance
   static Singleton()
   {
      instance = new Singleton();
   }

   // The property for retrieving the single instance
   public static Singleton Instance
   {
      get { return instance; }
   }

   // Private constructor: protects against direct instantiation
   private Singleton() { }
}
```

We have a **hidden (private) constructor** in order to limit external instantiations. We have a static variable, which holds **the only instance**. We initialize it only once in the **static constructor** of the class. The property for retrieving the single instance is usually called **Instance**.

The pattern can undergo many optimizations, such as the so called "**lazy initialization**" of the only variable, in order to save memory, but this is its classical form.

The Factory Method Design Pattern

Factory method is another very common design pattern. It is intended for **"producing" objects**. The instantiation of an object is not performed directly, but rather by the factory method. This allows the factory method to decide which specific instance to create from a family of classes implementing a common interface. The solution can depend on the environment, a parameter or some system setting.

The Factory Method Design Pattern – Example

Factory methods **encapsulate object creation**. This is useful if the creation process is very complicated – if it depends on settings in configuration files or input data by the user.

Suppose we have a class which contains graphics files (**png**, **jpeg**, **bmp**, etc.) and creates reduced size copies of them (the so called **thumbnails**). A variety of formats are supported, each represented by a class:

```
public class Thumbnail
{
  // …
}

public interface Image
{
  Thumbnail CreateThumbnail();
}

public class GifImage : Image
{
  public Thumbnail CreateThumbnail()
  {
    // … Create a GIF thumbnail here …
    return gifThumbnail;
  }
}

public class JpegImage : Image
```

```
{
   public Thumbnail CreateThumbnail()
   {
      // … Create a JPEG thumbnail here …
      return jpegThumbnail;
   }
}
```

Here is how the class holding an **album of images** looks like:

```
public class ImageCollection
{
   private IList<Image> images;

   public ImageCollection(IList<Image> images)
   {
      this.images = images;
   }

   public IList<Thumbnail> CreateThumbnails()
   {
      IList<Thumbnail> thumbnails =
         new List<Thumbnail>(images.Count);
      foreach (Image thumb in images)
      {
         thumbnails.Add(thumb.CreateThumbnail());
      }
      return thumbnails;
   }
}
```

The **client** of the program may require thumbnails of all images in the album:

```
public class Example
{
   static void Main()
   {
      IList<Image> images = new List<Image>();

      images.Add(new JpegImage());
      images.Add(new GifImage());

      ImageCollection imageRepository =
         new ImageCollection(images);
```

```
        Console.WriteLine(imageRepository.CreateThumbnails());
    }
}
```

Other Design Patterns

There are dozens of other well-known design patterns, but we are not going to discuss them. The more inquisitive readers can look up "Design Patterns" on the internet and learn what other design patterns, such as **Abstract Factory**, **Prototype**, **Adapter**, **Composite**, **Façade**, **Command**, **Observer**, **Iterator**, etc. serve for and how they are put into use. If you pursue .NET development more seriously, you will see for yourselves that the whole standard library (FCL) is built on the principles of OOP and the classic design patterns are very actively used.

Exercises

1. We are given a **school**. The school has classes of students. Each class has a set of **teachers**. Each teacher teaches a set of **courses**. The students have a name and unique number in the class. **Classes** have a unique text identifier. Teachers have names. Courses have a name, count of classes and count of exercises. The teachers as well as the students are people. Your task is to model the classes (in terms of OOP) along with their attributes and operations define the class hierarchy and create a class diagram with Visual Studio.

2. Define a class **Human** with properties "first name" and "last name". Define the class **Student** inheriting **Human**, which has the property "mark". Define the class **Worker** inheriting **Human** with the property "wage" and "hours worked". Implement a "calculate hourly wage" method, which calculates a worker’s hourly pay rate based on wage and hours worked. Write the corresponding constructors and encapsulate all data in properties.

3. Initialize an array of 10 students and sort them by mark in ascending order. Use the interface **System.IComparable<T>**.

4. Initialize an array of 10 workers and sort them by salary in descending order.

5. Define an abstract class **Shape** with abstract method **CalculateSurface()** and fields **width** and **height**. Define two additional classes for a **triangle** and a **rectangle**, which implement **CalculateSurface()**. This method has to return the areas of the rectangle (**height*width**) and the triangle (**height*width/2**). Define a class for a **circle** with an appropriate constructor, which initializes the two fields (**height** and **width**) with the same value (the radius) and implement the abstract method for calculating the area. Create an array of different shapes and calculate the area of each shape in another array.

6. Implement the following classes: **Dog**, **Frog**, **Cat**, **Kitten** and **Tomcat**. All of them are animals (**Animal**). Animals are characterized by **age**, **name** and **gender**. Each animal makes a sound (use a virtual method in the **Animal** class). Create an array of different animals and print on the console their name, age and the corresponding sound each one makes.

7. Using Visual Studio generate the **class diagrams** of the classes from the previous task with it.

8. A **bank** holds different **types of accounts** for its customers: **deposit** accounts, **loan** accounts and **mortgage** accounts. Customers can be **individuals** or **companies**. All accounts have a customer, balance and interest rate (monthly based). **Deposit accounts** allow depositing and withdrawing of money. **Loan and mortgage accounts** allow only depositing. All accounts can calculate their interest for a given period (in months). In the general case, it is calculated as follows: **number_of_months * interest_rate**. **Loan accounts** have no interest rate during the first 3 months if held by individuals and during the first 2 months if held by a company. **Deposit accounts** have no interest rate if their balance is positive and less than 1000. **Mortgage accounts** have ½ the interest rate during the first 12 months for companies and no interest rate during the first 6 months for individuals. Your task is to write an object-oriented model of the bank system. You must identify the classes, interfaces, base classes and abstract actions and implement the interest calculation functionality.

9. Read about the **Abstract Factory** design pattern and implement it in C#.

Solutions and Guidelines

1. The task is trivial. Just follow the problem description and write the code.

2. The task is trivial. Just follow the problem description and write the code.

3. Implement **IComparable<T>** in **Student** and then sort the array.

4. This problem is like the previous one.

5. Just implement the classes as described in the problem description.

6. Printing information can be implemented in the virtual method **System.Object.ToString()**. In order to print the content of an array of animals, you can use a **foreach** loop.

7. If you have the full version of **Visual Studio**, just use "**Add New Item**" → "**Class Diagram**". Class diagrams are not supported in VS Express Edition. In this case you can find some other UML tool (see http://en.wikipedia.org/wiki/List_of_UML_tools).

8. Use abstract class **Account** with abstract method **CalculateInterest(…)**.

9. You can read about the "**abstract factory**" design pattern in Wikipedia: http://en.wikipedia.org/wiki/Abstract_factory_pattern.

Chapter 21. High-Quality Programming Code

In This Chapter

In this chapter we review the basic rules and **recommendations for writing quality program code**. We pay attention to **naming** the identifiers in the program (variables, methods, parameters, classes, etc.), **formatting** and **code organization** rules, good practices for **composing methods**, and principles for writing **quality documentation**. We describe the official "Design Guidelines for Developing Class Libraries for .NET" from Microsoft. In the meantime we explain how the programming environment can automate operations such as **code formatting** and **refactoring**.

This chapter is a kind of continuation of the previous one – "Object-Oriented Programming Principles". The reader is expected to be familiar with the basic OOP principles: **abstraction**, **inheritance**, **polymorphism**, **encapsulation** and **exception handling**. Those do greatly affect the quality of the code.

Why Is Code Quality Important?

Let's examine the following code:

```
static void Main()
{
  int value=010, i=5, w;
  switch(value){case 10:w=5;Console.WriteLine(w);break;case
9:i=0;break;
          case 8:Console.WriteLine("8 ");break;
     default:Console.WriteLine("def ");{
          Console.WriteLine("hoho "); }
     for (int k = 0; k < i; k++, Console.WriteLine(k -
'f'));break;} { Console.WriteLine("loop!"); }
}
```

Are you able to comprehend **what this code does** in a short glance? Does it do its job correctly, does it contain any errors?

What Does Quality Programming Code Mean?

The quality of a program encompasses two aspects: the quality perceived by the user (called **external quality**), and the quality in regard to the internal organization (called **internal quality**).

The **external quality** is largely determined by the operational **correctness** of the particular program (absence of defects). Things like **usability** and intuitiveness of the user interface (UI) do greatly influence the external quality as well. **Performance**, a term which includes operational speed, memory usage and resource utilization, also plays in the equation, whenever these things matter.

Internal quality, on the other hand, is determined by **how well the program is built**. It depends on whether the employed design and architecture are suitable and sufficiently simple, and whether it is **easy to make a change** or to add new functionality (**maintainability**). The comprehensibility of the implementation and the **readability of the code** are vital as well. In general, internal quality mostly has to do with the code of the program and its internal work.

Characteristics of Quality Code

Quality code is **easy to read** and understand. It is **maintained easily** and straightforwardly. It must withstand any kind of input without breaking or behaving strangely, and be **well tested**. The **design** and the **architecture** must be suitable and not over-engineered. Documentation should be at a decent level, or at least the code should be **self-documenting**. **Formatting** should be adequately chosen and applied **consistently** throughout the whole project.

At all levels (modules, classes, methods) there should be a strong relation and a high focus of the responsibilities (**strong cohesion**) – that means, a piece of code should only **do one particular thing**.

Functional independence (or more precisely, **loose coupling**) between modules, classes and methods is crucially important. Suitable and consistent **naming** of all program identifiers is a must. **Documentation** should be embedded in the code itself.

Why Should We Write Quality Code?

Let’s have a look again at our example:

```
static void Main()
{
  int value=010, i=5, w;
  switch(value){case 10:w=5;Console.WriteLine(w);break;case
9:i=0;break;
          case 8:Console.WriteLine("8 ");break;
```

```
    default:Console.WriteLine("def ");{
         Console.WriteLine("hoho ");}
    for (int k = 0; k < i; k++, Console.WriteLine(k -
'f'));break;} { Console.WriteLine("loop!"); }
}
```

Can you tell whether this code **compiles without errors**? Can you tell **what it does** just by glancing at it? Can you **add new functionality** and be sure that you will not break it up? Can you tell what the purpose of the **k** or the **w** variable is?

Visual Studio has an option for **automatic code formatting**. If the above code is put there and that option is invoked (via the keyboard combination [Ctrl+K, Ctrl+F]) it will be reformatted and will look completely differently. Unfortunately, the purpose of the variables will still remain unclear, but at least it should become obvious where each block ends:

```
static void Main()
{
  int value = 010, i = 5, w;
  switch (value)
  {
    case 10: w = 5; Console.WriteLine(w); break;
    case 9: i = 0; break;
    case 8: Console.WriteLine("8 "); break;
    default: Console.WriteLine("def ");
      {
        Console.WriteLine("hoho ");
      }
      for (int k = 0; k < i; k++, Console.WriteLine(k - 'f'));
      break;
  } { Console.WriteLine("loop!"); }
}
```

If everyone was writing code as in the above example, it would not be possible to create big and serious software projects, because they are written by large teams of software engineers. If every team member's code was like that, **no one would ever be able to understand** how the other members' code works (and whether it works at all), and hardly anyone could even understand his / her own code.

Over the time, a serious amount of **good practices** have emerged and a lot of experience has been gained for writing quality code, to enable each programmer to **understand and maintain his colleagues' code**. These practices endorse a variety of rules and recommendations for code **formatting**, identifier **naming** and proper **program structure**, all of which make writing software easier. Consistent and quality code is especially helpful

when changing and **maintaining a program**. Quality code is flexible and stable. Because it is self-documenting and intuitive, its intentions become clear at a first sight. Quality code is **easy to reuse** because it does just one thing (**strong cohesion**), but does it well, depending on a minimal amount of other components (**loose coupling**), using only their public interfaces. As an end result, quality code saves time and labor, and makes the produced software more valuable.

Some programmers consider quality code as being overly simple. They tend to think that it limits their opportunity to demonstrate their knowledge. That is the reason why they write **code that is hard to read**, and for using features of the language which are unpopular or poorly documented. They squeeze functions on a single line. This is an **entirely wrong practice**.

Coding Conventions

Before continue with the recommendations on writing quality code, we should talk a bit about coding conventions. A **coding convention** is a **set of rules for writing code**, used within the boundaries of a particular organization or a project. It can include naming and formatting rules, and rules for logical composition. One such rule would recommend that class names start with a capital letter while variable names start with a lowercase letter. Another rule may state that the opening curly bracket preceding a block of statements should be on the same line, rather than on a new line.

Inconsistent usage of a single convention is worse and more dangerous than not having a convention at all.

Conventions started to emerge in big and serious projects, where a large number of programmers had each been writing in their own style and everyone was adhering to their own (if any) rules. This was making the code harder to read and has forced project managers to introduce **written rules**. Later, the best coding conventions gained popularity and have become a de facto standard.

Microsoft provides an **official coding convention** called **.NET Framework Guidelines and Best Practices** for .NET 4.5 (http://msdn.microsoft.com/en-us/library/ms184412.aspx).

Since then, this coding convention has gained significant popularity and has become very widespread. The naming and formatting rules presented here are in sync with the above convention from Microsoft.

Large organizations adhere to **strict conventions**. Among separate teams, conventions may differ, however. Most team leaders choose to stick with the official convention of Microsoft, and they eventually extend it when necessary.

Code quality is not just a set of rules, which must be adhered to; it is rather a way of thinking.

Managing Complexity

The management of complexity plays a central role when writing software. The main objective is to **reduce the complexity that each member has to deal with** at a certain moment. This way the brain of each of the members is burdened with less stuff to think about.

The **complexity management** starts from the architecture and the design. Each and every module (or rather, each autonomous code unit) should be designed with reducing complexity in mind.

Good practices should be applied at all levels – classes, methods, member variables, naming, operators, error handling, formatting, comments, etc. They transform a lot of the decisions about the code in a strictly-defined set of rules, which enables a developer to think about one thing less while reading and writing code.

The complexity management can be approached in another way: it is especially helpful for a developer to be able to **abstract himself away from the big picture while writing a small piece of code**. For that to be possible, the piece of code should have very clear boundaries, which are in-tact with the big picture. The old Roman rule "Divide and conquer" still applies when complexity is concerned.

The rules we are talking about later on are directed exactly towards eliminating complexity while working on a single, small piece of the system.

Identifier Naming

Identifiers are the **names** of classes, interfaces, structures, enumerations, properties, methods, parameters and variables. In C# and in many other languages, **names are chosen by the developer**. Names should not be random. They should be composed in such a way that they carry **meaningful information about their purpose** and their role in the code. This makes the code easier to read.

When naming an identifier, it is good to ask yourself these questions: What does this class do? What is the purpose of this variable? What is that method being used for? What information does this parameter hold?

Some **good names** are: **FactorialCalculator**, **studentsCount**, **Math.PI**, **configFileName**, **CreateReport**.

Some **bad names** are: **k**, **k2**, **k3**, **junk**, **f33**, **KJJ**, **button1**, **variable**, **temp**, **tmp**, **temp_var**, **something**, **someValue**.

It is especially bad to have a class or a method called **Problem12**. Some beginner programmers would give such a name to their solution of Problem 12 from the exercises. What will the name **Problem12** tell you in a week or a month? If the problem is about finding a path in a labyrinth, name it **PathInLabyrinth**. Three months later you may encounter a similar problem and you will be able to find the labyrinth problem. How would you find it if you

have named it inappropriately? Do not give a name that contains digits – this is an indication for bad naming.

	The name of an identifier should describe its purpose. The solution of problem 12 from the exercises should NOT be called Problem12. That is a huge mistake!

Avoid Abbreviations

Abbreviations should be avoided because they can be confusing. What does the class name **GrBxPnl** tell you? Isn't **GroupBoxPanel** clearer? Exceptions are made for acronyms, which are more popular than their full form, for example HTML or URL. In that sense, **HTMLParser** is recommended over the excessive long name **HyperTextMarkupLanguageParser**.

Use English

One of the most basic rules is to **always use English**. Would you be able to understand the code of a foreigner who names variables and methods in his own language? The one and only human language, which all programmers should know, is English.

	English is a de facto standard in writing software. Always use English for naming the identifiers in the code (variables, methods, classes, etc.). Use English for comments as well.

Let's see how we pick appropriate identifiers in different cases.

Consistency in Naming

Naming should be **consistent**. What does this mean?

In a group of methods called **LoadFile()**, **LoadSettings()**, **LoadFont()**, **LoadImageFromFile()** and **LoadLibrary()** it is inappropriate to have a method **ReadTextFile()**. The word **Read** is not consistent with **Load**.

Opposite activities should be **symmetrically named** (you should be able to guess the name of the opposite activity just by knowing the complementing one): **LoadLibrary()** goes with **UnloadLibrary()**, but does not go with **FreeHandle()**. **OpenFile()** goes with **CloseFile()**, but does not go with **DeallocateResource()**. It is unnatural to have **AssignName** next to a **GetName** / **SetName** pair.

Notice that in .NET Framework class library, big groups of classes have **consistent naming**: collections (the namespace and all classes use the words like **Collection** and **List**, and never their synonyms), streams are always **Streams**, etc.

Use consistent names: use the same words for the same situations, do not use synonyms. Name opposite things symmetrically.

Names of Classes, Interfaces and Other Types

From the chapter "Principles of Object-Oriented Programming" we know that classes describe real-world objects. Class names should consist of a **noun** (denominative or substantive) and possibly a **number of adjectives** (before or after the noun). For example, a class describing an African lion should be called **AfricanLion**.

The recommended casing of the letters (small / capital letters) for naming types in C# is **Pascal Case**: the first letter of every word in the name is always uppercase and the rest of the letters are lowercase. This way it is easier to read the identifier's name (compare the lowercase name **idatagridcolumnstyleeditingnotificationservice** to its Pascal Case version **IDataGridColumnStyleEditingNotificationService**). The latter is the public class with probably the longest name in the .NET Framework (46 characters, from **System.Windows.Forms**).

Let's give a few more examples. We are to write a class, which finds the prime numbers in a given range. A **good** name for that class is **PrimeNumbers** or **PrimeNumbersFinder**, or maybe **PrimeNumbersScanner**. **Bad names** would be **FindPrimeNumber** (a verb should not be used in the name of a class) or **Numbers** (it is not clear what the numbers are and what we are doing with them) or **Prime** (a class name should not be an adjective).

How Long Should Class Names Be?

In the common case, class names **should not exceed 20 characters**, but sometimes this rule is not adhered to if a real-world object is described which contains numerous longer words. As we saw above, it is possible to have a class with a name that is 46 characters long. Although the name is long, it is very clear what this class does. Because of this, the recommendation for class names being less than 20 characters is only advisory, **not mandatory**. If you are about to choose between a class name that is short and clear and another one which is longer and as clear as the short one, prefer the short name.

A bad advice would be to abbreviate names in order to keep them short. Are the following class names clear enough: **CustSuppNotifSrvc**, **FNFException**? Obviously they are not easy readable. Names like **FileNotFoundException** and **CustomerSupportNotificationService** are much clearer, although being longer.

Naming Interfaces and Other Types

Interface names should follow the same convention as class names: they are written in **Pascal Case** and consist of a **noun and possibly a few**

adjectives. In order to distinguish them from the rest of the types, the convention suggests prefixing them with an "**I**".

Some **good** examples are: **IEnumerable**, **IFormattable**, **IDataReader**, **IList**, **IHttpModule**, **ICommandExecutor**.

Bad examples would be: **List**, **FindUsers**, **IFast**, **IMemoryOptimize**, **Optimizer**, **FastFindInDatabase**, **CheckBox**.

In .NET there is one more notation for naming interfaces: naming them so that they end in "**able**": **ICloneable**, **IEnumerable**, **IFormattable**. These are interfaces that most often augment the basic role of an object. Most interfaces, however, do not follow this notation, such as the **IList** and **ICollection** interfaces.

Names of Enumeration Types

A few formats are allowed for **naming enumerations**: **[Noun]** or **[Verb]** or **[Adjective]**. Names can be in singular or plural form. Every member of an enumerated type should be named in the same manner. The below examples show **correctly named enumerations**:

```
enum Days
{
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};

enum Color
{
  Black, Red, Green, Blue, Yellow, Orange, Pink, Gray, White
};
```

Attribute Names

Attribute names in C# should be suffixed with **Attribute**. For example: **WebServiceAttribute**. Attributes are special annotations (metadata) for a class / method or other piece of code which specify a special instruction for the compiler or the runtime. For more information see the documentation in MSDN: http://msdn.microsoft.com/en-us/library/z0w1kczw(v=vs.110).aspx.

Exception Names

The convention for naming exception classes suggests that exceptions end with **Exception**. The name should be informative and Pascal case should be used just like when naming classes. A **good example** of correctly named exception class would be **FileNotFoundException**. A bad example for exception class is **FileNotFoundError**.

Delegate Names

Delegates in C# and .NET Framework should be suffixed with **Delegate** or **EventHandler**. Thus **DownloadFinishedDelegate** would be a good example while **WakeUpNotification** would not adhere to the convention. A **delegate** is a data type which holds a reference to method with compatible signature. For more information about delegates see the official documentation in MSDN: http://msdn.microsoft.com/en-us/library/ms173171(v=vs.110).aspx.

Naming Namespaces

Namespaces, covered in details in the "Creating and Using Objects" chapter, should use Pascal Case like class names. The following forms are preferable:

- **Company.Product.Component…**
- **Product.Component…**

Good example for naming a namespace is: **Telerik.WinControls.GridView**.

Bad examples for naming namespaces are: **Classes**, **TELERIK.CONSTANTS** and **Telerik_WinControlsGridView**.

Assembly Names

Assembly names should match the name of **the base namespace which they hold**. **Good** examples of correctly named assemblies are:

- **Telerik.WinControls.GridView.dll**
- **Oracle.DataAccess.dll**
- **Interop.CAPICOM.dll**

Improper (**bad**) assembly names are the following:

- **Telerik_WinControlsGridView.dll**
- **OracleDataAccess.dll**
- **Oracle.dll**

Method Names

Method names should be **PascalCase**, e.g. each separate word starts with an uppercase letter.

Method names should be constructed according to the following pattern: **<verb> + <object>**, for example **PrintReport()**, **LoadSettings()** or **SetUserName()**. The object can be a noun or a noun and an adjective: **ShowAnswer()**, **ConnectToRandomTorrentServer()** or **FindMaxValue()**.

A method name should address **what that method does**. If you are not able to come up with a good name, you most probably have to review the method itself and whether it is decently written.

Some **bad** examples for method names are: **DoWork()** (what kind of work?), **Printer()** (no verb), **Find2()** (why not **Find7()** then?), **ChkErr()** (abbreviations are not recommended), **NextPosition()** (no verb).

Sometimes a **single verb** is a good name for a method, as long as it becomes clear what the particular method does and what objects it operates on. For example, within a **Task** class, the methods **Start()**, **Stop()** and **Cancel()** are well-named because it is clear that they start, stop or cancel a task. In other cases a single verb is inappropriate. For example, within an **Utils** class, methods called **Evaluate()**, **Create()** and **Stop()** are inadequate because the **context** is not entirely clear.

Methods that Return a Value

Names of methods that return a value should **describe the returned value** in some way, e.g. **GetNumberOfProcessors()**, **FindMinPath()**, **GetPrice()**, **GetRowsCount()**, **CreateNewInstance()**.

Bad examples for names of methods that return a value are the following: **ShowReport()** (it is not clear what the method returns), **Value()** (should be either **GetValue()** or **HasValue()**), **Student()** (no verb), **Empty()** (should be **IsEmpty()**).

Whenever a value is returned, the **measuring unit** should be clear: **MeasureFontInPixels(…)**, instead of **MeasureFont(…)**.

Single Purpose of a Method

A method that does more than one thing is hard to be appropriately named: how would you call a method that does an annual income report, downloads software updates from the web and scans the system for viruses? Maybe **CreateAnnualIncomesReportDownloadUpdatesAndScanForViruses**?

Methods should have one purpose only, solving only one task, not multiple tasks at the same time!

Methods solving multiple tasks (**weak cohesion**) cannot and should not be named properly. They must be **refactored**.

Cohesion and Naming

A name should describe **everything that the method does**. If a suitable name cannot be found, it most probably means that the cohesion is weak, i.e. the method does many things and should be split up into separate methods.

Here is an example: we have a method that sends an e-mail, prints a report and calculates the distance between two points in 3D Euclidean space. How would you call it? Maybe **SendEmailAndPrintReportAndCalc3DDistance()**? It is obvious that **something is wrong with this method** – we should refactor it instead of striving to find a better name. It is even worse if that method is simply called **SendEmail()**. This way we are misleading other

programmers that this method only sends email, while in reality it does many other things. The last is very, very bad practice.

Naming a method misleadingly is even worse than naming it `method1()`. If a method calculates a cosine and we name it `sqrt()`, we will likely enrage other colleagues that are willing to use our code.

How Long Should Method Names Be?

The same recommendations apply here as for classes – you should not abbreviate unless it is clear. The **names should be meaningful** and this is more important than its length. If the method name is too long (e.g. more than **50 characters**), check whether it does a single task.

Good method names are: **`LoadCustomerSupportNotificationService()`**, **`Math.Sqrt()`**, **`CreateMonthlyAndAnnualIncomesReport()`**.

Bad method names are **`LoadCustSuppSrvc()`**, **`CreateMonthIncReport()`**.

Method Parameters

Parameters should be named in the following form: **[Noun]** or **[Adjective] + [Noun]**. Every word of the name should start with an uppercase letter, except for the first word. This notation is called **`camelCase`**. As with any other code element, parameter naming should be meaningful and should carry useful information.

Good examples of parameter names are the following: **`firstName`**, **`report`**, **`fontSizeInPixels`**, **`speedKmH`**, **`font`**, **`usersList`**.

Bad examples of parameter names are: **`p`**, **`p1`**, **`p2`**, **`populate`**, **`LastName`**, **`last_name`**, **`convertImage`**.

Property Names

Property names start with an uppercase letter (**`PascalCase`**, like methods), but do not contain a verb (like variables). A property name consists of an **[Adjective] + [Noun]** or just **[Noun]**.

In the presence of a property called **`X`**, it is not a good practice to have a method called **`GetX()`** – it will be confusing.

If the property is of type enumeration, you could think about naming the property like the enumeration type itself. In the presence of an enumeration called **`CacheLevel`**, the property would as well be called **`CacheLevel`**.

Using **the same name for the property and its type** is allowed and is usual in .NET Framework class library. For example the property **`Cursor`** of the class **`Button`** in Windows Forms is of type **`Cursor`**.

Variable Names

Variable names (local variables in a method) and member-variables (fields in a class) should adhere to the **camelCase** notation, according to Microsoft.

Variables should have a **good name**, as all other identifiers in the code should. A good variable name clearly and precisely describes the object that the variable holds. **Good** variable names are: **account**, **blockSize** and **customerDiscount**. Bad names are: **r18pq**, **__hip**, **rcfd**, **val1**, **val2**.

A name should **address the problem that the variable solves**. It should answer the question "What?", not "How?". In this sense, **good** names are **employeeSalary**, **employees**. **Bad** names are the ones that are irrelevant to the solved problem: **myArray**, **customerFile**, **customerHashTable**.

Prefer names from the business domain in which the software operates, not from the technical names that come from the programming language: use CompanyNames rather than StringArray.

The **optimal length of a variable name is from 10 to 16 characters**. The length of the name depends on the scope – variables with wider scope and a longer lifetime should have a more descriptive name:

```
protected Account[] customerAccounts;
```

Variables with a narrower scope and a shorter lifetime could have shorter length:

```
for (int i=0; i < customers.Length; i++) { … }
```

Variable names should be instantly **understandable**. Because of this it is not a good idea to remove vowels from the name in order to abbreviate it – **btnDfltSvRzlts** is **not** quite understandable.

The most important thing is: whatever naming rules are chosen for variables, they should be applied **consistently** throughout the code – in all the modules of the project and by the whole team. An inconsistently applied rule is worse than not having a rule at all.

Names of Boolean Identifiers

Parameters, properties and variables can be of a Boolean type. In this point we describe the specifics of these identifiers.

Their names should be a prerequisite for either truth or falsehood. For example, names like **canRead**, **available**, **isOpen** and **valid** are **good**. Examples of **inadequate** names for Boolean variables are: **student**, **read**, **reader**.

It would be useful if Boolean identifiers start with **is**, **has** or **can** (with an uppercase letter for properties), but only if this adds for clarity.

Negated names should not be used (avoid prefixing with "**not**"), because the following oddities may occur:

```
if (! notFound) { … }
```

Good examples: **configFileLoaded**, **hasPendingPayment**, **customerFound**, **validAddress**, **positiveBalance**, **isPrime**.

Bad examples: **notFound**, **fsafdashghg**, **run**, **programStop**, **player**, **list**, **findCustomerById**, **isUnsuccessfull**.

Named Constants

Like we already know from the chapter "Defining Classes" constants in C# are something like static immutable variables and are defined as follows:

```
public struct Int32
{
  public const int MaxValue = 2147483647;
}
```

Names of constants should be written in **Pascal Case** or entirely in uppercase, with underscores between words (**ALL_CAPS**):

```
public static class Math
{
  public const double PI = 3.14159265359;
  public const double GoldenRatio = 1.61803398875;
}
```

Named constants should **clearly describe** what **the purpose** of the particular number, string or whatever value is, rather than the value itself. A constant named **number314159** is useless and confusing.

The official recommendation from Microsoft for naming the constants (**const** and **readonly** identifiers) is to use Pascal Case but some developers prefer the ALL_CAPS style which is widely used in C++ and Java.

Naming of Specific Data Types

Names of variables used as **counters** are recommended to contain a word that specifies that, for example **usersCount**, **rolesCount**, **filesCount**.

Variables that represent the **state** of an object should be named accordingly. A few examples: **threadState**, **transactionState**.

Temporary variables should most often have common and short names, which make obvious that they are temporary, with a very short lifetime. Good examples are **index**, **oldValue**, **count**. Inappropriate names are **a**, **aa**, **tmpvar1**, **tmpvar2**. Although using names like **tmp** and **temp** is acceptable it is better to choose more meaningful name like **oldValue** and **lastIndex**.

Naming by Prefixing or Suffixing

Prefix and **suffix** naming conventions do exist in older languages such as C. A very popular notation during many years has been the **Hungarian notation**. Hungarian notation is a prefix naming notation in which every variable comes with a prefix that indicates its type and purpose. For example, in Win32 API, the name **lpcstrUserName** would mean a variable that is a pointer to an array of characters, which ends in 0, and is interpreted as a non-Unicode string.

In C#, .NET Framework, Java and all modern programming languages, similar conventions have never gained popularity because the development environments are able to show the type of any variable. **Do not use Hungarian notation** in C#! Exceptions are made by some graphics libraries, to a certain extent.

Code Formatting

Formatting, along with naming is one of the most basic prerequisites for readable code. Without **proper formatting**, the code is not going to be **readable**, whatever rules for naming and code structuring are chosen.

Formatting has two objectives: **easier to read code**, and, as a consequence – code that is **easy to maintain**. Formatting that makes the code harder to read is not good. Every aspect of formatting (indentation, empty lines, alignment, etc.) can provide benefits as well as cause harm. Formatting should **follow the logical structure of the program** so that the logical understanding is supported.

The formatting of a program should represent its logical structure. All formatting rules are introduced towards improving code readability by exposing its logical structure.

In Visual Studio, the code can be **automatically formatted** with the [Ctrl+K, Ctrl+F] key combination. Different standards can be applied whenever auto formatting is performed – the Microsoft conventions as well as user-defined standards are available. Try it yourself: select a piece of code and press [Ctrl+K] followed by [Ctrl+F].

Now we are going to review the formatting rules according to the coding convention of Microsoft for C#.

Why Does Code Need Formatting?

First let's look at the below example:

```
public   const    string                   FILE_NAME
="example.bin"  ;  static void Main   (             ){
FileStream   fs=     new FileStream(FILE_NAME,FileMode
.   CreateNew)   // Create the writer      for data  .
;BinaryWriter w=new BinaryWriter     (    fs       );//
Write data to                             Test.data.
for(  int i=0;i<11;i++){w.Write((int)i);}w   .Close();
fs    .    Close   (   ) // Create the reader    for data.
;fs=new FileStream(FILE_NAME,FileMode.            Open
,  FileAccess.Read)     ;BinaryReader                r
= new BinaryReader(fs);  // Read data from  Test.data.
 for (int i = 0; i < 11; i++){ Console      .WriteLine
(r.ReadInt32                                       ())
;}r       .    Close   (   );  fs .  Close  (  )  ;  }
```

Is that enough as an answer?

Block Formatting

Blocks are surrounded by "**{**" and "**}**". In C# they should be on **separate lines** (unlike in Java and JavaScript). The contents of the block should be indented to the right by a single tab:

```
if (some condition)
{
  // Block contents indented by a single [Tab]
  // Don't use spaces for indentation
}
```

This rule applies for namespaces, classes, methods, conditional statements, loops, etc.

Nested blocks are indented additionally. The body of the class here is indented relative to the body of the namespace, and the body of the method is indented additionally, as well as the conditional statement:

```
namespace Chapter_21_Quality_Code
{
  public class IndentationExample
  {
    private int Zero()
    {
      if (true)
```

```
            {
                return 0;
            }
        }
    }
}
```

Rules for Formatting a Method

According to the Microsoft's coding convention, some formatting rules when declaring methods should be adhered to.

Formatting Multiple Method Declarations

Whenever a class has more than one method, their declarations should be separated by an **empty line**:

IndentationExample.cs

```
public class IndentationExample
{

    public static void FirstMethod()
    {
        // …
    } // One blank line follows

    public static void SecondMethod()
    {
        // …
    }
}
```

How to Put Parentheses?

The Microsoft coding convention suggests that a **space should be put** between a keyword (**for**, **while**, **if**, **switch**) and an opening parenthesis:

```
while (!EOF)
{
    // … Code …
}
```

This is made for the keywords to stand out.

Next to a method name and before an opening parenthesis, **no whitespace** should be present:

```
public void CalculateCircumference(int radius)
{
   return 2 * Math.PI * radius;
}
```

In this line of thought, between the name of the method and the opening parenthesis "**(**" there **should not be any whitespace** (spaces, tabs etc.):

```
public static void PrintLogo()
{
   // … Code …
}
```

Formatting the Parameter List of Methods: Space after Commas

When a method has many parameters, we should put a space between the previous comma and the type of the next parameter, but **not before the comma**:

```
public void CalcDistance(Point startPoint, Point endPoint)
```

Similarly, the same rule is applied when calling a method with more than one parameter. Before the arguments preceded by a comma, a **space** should be put:

```
DoSmth(1, 2, 3);
```

Rules for Formatting of Types

When classes, interfaces, structures and enumerations are created, a few recommendations should be followed for formatting the code inside.

Rules for Ordering the Contents of a Class

As we know, the class name is declared on the first line, preceded by the **class** keyword:

```
public class Dog
{
```

Constants follow next. They should be ordered according to their access modifier – **public** constants are first, then **protected** and then **private**:

```
   // Static variables
   public const string SPECIES = "Canis Lupus Familiaris";
```

Then follow the non-static fields. Like static fields, those labeled **public** are first, then **protected** and finally **private** fields follow:

```
   // Instance variables
   private int age;
```

After non-static class fields, constructor declarations follow:

```
   // Constructors
   public Dog(string name, int age)
   {
      this.Name = name;
      this.age = age;
   }
```

After the constructors, properties are declared:

```
   // Properties
   public string Name { get; set; }
```

Finally, after the properties, the methods are declared. It is recommended that methods are grouped by functionality, not by access level or scope. For example, a method with a **private** access modifier could easily be between two methods with a **public** modifier in order to make reading and understanding the code easier. We end by putting a curly bracket for the end of the class:

```
   // Methods
   public void Breath()
   {
      // TODO: breathing process
   }

   public void Bark()
   {
      Console.WriteLine("wow-wow");
   }
}
```

Formatting Rules for Loops and Conditional Statements

Formatting of loops and conditional statements follows the same rules as methods and classes. The body of a conditional statement or a loop is **always put in a block** beginning with "**{**" and ending with "**}**". The opening bracket is always on a new line, immediately after the condition of the loop or the

conditional statement. The body of a loop or a conditional statement is always **indented** to the right by a single tabulation. If the condition is long and does not fit at a single line, it is carried over on a new line and then indented to the right by two tabs. Here is an example of a correctly formatted loop and a conditional statement:

```
static void Main()
{
	Dictionary<int, string> bulgarianNumbersHashtable =
		new Dictionary<int, string>();
	bulgarianNumbersHashtable.Add(1, "one");
	bulgarianNumbersHashtable.Add(2, "two");
	bulgarianNumbersHashtable.Add(3, "three");

	foreach (KeyValuePair<int, string> pair in
		bulgarianNumbersHashtable.ToArray())
	{
		Console.WriteLine("Pair: [{0},{1}]", pair.Key, pair.Value);
	}
}
```

It is especially **wrong** to indent the body of a loop or a conditional statement as follows:

```
foreach (Student s in students) {
		Console.WriteLine(s.Name);
		Console.WriteLine(s.Age);
}
```

Usage of Empty Lines

It is very common for beginner programmers to insert **empty lines** in a chaotic manner. Really, when new lines do not harm, why shouldn't we put them wherever we want and why should we remove them since they do not affect the meaning of the code? The reason is very simple: **empty lines** are used for **separating parts of the program**, which are not **logically connected**, much like new lines separate the end and the beginning of two paragraphs.

Empty lines are used to **separate two methods**, to separate a group of member-variables from another group of member-variables with a different logical task, for separating a group of related statements from another group of related statements.

Here is a **bad example** of two methods in which empty lines are used **inappropriately** and that hinders code readability:

```
public static void PrintList(IList<int> list)
{
  Console.Write("{ ");
  foreach (int item in list)
  {
    Console.Write(item);

    Console.Write(" ");


  }
  Console.WriteLine("}");
}
static void Main()
{
  IList<int> firstList = new List<int>();
  firstList.Add(1);

  firstList.Add(2);
  firstList.Add(3);
  firstList.Add(4);
  firstList.Add(5);
  Console.Write("firstList = ");
  PrintList(firstList);
  List<int> secondList = new List<int>();
  secondList.Add(2);

  secondList.Add(4);
  secondList.Add(6);
  Console.Write("secondList = ");
  PrintList(secondList);
  List<int> unionList = new List<int>();
  unionList.AddRange(firstList);
  Console.Write("union = ");

  PrintList(unionList);
}
```

You see that the empty lines do not **represent the logical structure of the program**, and that is why they break the main rule in formatting.

If we reformat the program so that **empty lines are properly used** to separate logically related groups of statements, we will come up with much **more readable code**:

```
public static void PrintList(IList<int> list)
{
  Console.Write("{ ");
  foreach (int item in list)
  {
    Console.Write(item);
    Console.Write(" ");
  }
  Console.WriteLine("}");
}

static void Main()
{
  IList<int> firstList = new List<int>();
  firstList.Add(1);
  firstList.Add(2);
  firstList.Add(3);
  firstList.Add(4);
  Console.Write("firstList = ");
  PrintList(firstList);

  List<int> secondList = new List<int>();
  secondList.Add(2);
  secondList.Add(4);
  secondList.Add(6);
  Console.Write("secondList = ");
  PrintList(secondList);

  List<int> unionList = new List<int>();
  unionList.AddRange(firstList);
  Console.Write("union = ");
  PrintList(unionList);
}
```

Rules for Moving to the Next Line and Alignment

When a line is longer, split it up into two or more lines and indent the lines after the first one by a **single tab**:

```
Dictionary<int, string> egyptianNumbersHashtable =
  new Dictionary<int, string>();
```

It is **wrong to align similar statements** according to the longest of them, since that can obstruct the maintenance of the code:

```
DateTime         date     = DateTime.Now.Date;
int              count    = 0;
Student          student  = new Strudent();
List<Student>    students = new List<Student>();
```

Or

```
matrix[x, y]                  == 0;
matrix[x + 1, y + 1]          == 0;
matrix[2 * x + y, 2 * y + x]  == 0;
matrix[x * y, x * y]          == 0;
```

It is **wrong** to align arguments to the right, based on the opening parenthesis of a method call:

```
Console.WriteLine("word '{0}' is seen {1} times in the text",
                  wordEntry.Key,
                  wordEntry.Value);
```

The above code should be **properly formatted** as follows (this is not the only proper way, though):

```
Console.WriteLine(
   "word '{0}' is seen {1} times in the text",
   wordEntry.Key,
   wordEntry.Value);
```

High-Quality Classes

Let's now discuss the classes and the **best practices** about using efficiently classes when writing high-quality code.

Software Design

When a system is designed, separate **subtasks** are often divided into separate **modules** or **subsystems**. The task that each one solves must be clearly defined. The relationships between the modules should be decided in advance, not on the go.

In the previous chapter we explained OOP and we showed how object-oriented modeling can be used to **define classes** of the real actors in the domain of the solved problem. We mentioned **design patterns** as well.

Good software design has **minimal complexity** and is **easy to understand**. It is **maintained easily** and changes are incorporated straightforwardly (see the "Spaghetti Code" section in the previous chapter). Every program element (method, class, module) is logically connected internally (**strong cohesion**),

functionally-independent and minimally tied to the other modules (**loose coupling**). Well-designed code is **easily reused**.

Object-Oriented Programming (OOP)

When creating quality classes, the main rules stem from the four main OO principles: **abstraction**, **inheritance**, **encapsulation** and **polymorphism**.

Abstraction

A few basic rules:

- Public properties of a class should have the same level of abstraction.
- The interface of a class should be simple and clear.
- A class should describe only one thing.
- A class should hide its internal implementation.

Code is developed and changes and evolves over time. In spite of the evolution of classes, their interfaces should remain in-tact. A **bad practice** of a class having **inconsistent interface** is shown below:

```
class Employee
{
  public string firstName;
  public string lastName;
  …
  public SqlCommand FindByPrimaryKeySqlCommand(int id);
}
```

The latter method is incompatible with the level of abstraction at which **`Employee`** works. The user of this class should not be aware at all that a database is used internally.

Inheritance

Do not hide methods in derived classes:

```
public class Timer
{
  public void Start() { … }
}

public class AtomTimer : Timer
{
  public void Start() { … }
}
```

The method in the derived class **hides** the base (original) implementation. This is **not recommended**. If, in a rare case, this is desired and necessary, the keyword **new** should be used.

Move common methods, data and behavior as high as possible in the inheritance tree. This way, functionality is less likely to be duplicated and will be accessible to a wider audience.

If you have a class with a **single successor only**, consider this suspicious. That level of abstraction is probably unnecessary. A suspicious method would be one that re-implements a base method, but does nothing more than the corresponding base method.

Deep inheritance with **more than 6 levels** is hard for tracing, debugging and maintaining, and is **not recommended**. In a derived class, use member-variables through properties, rather than directly.

The example below demonstrates **wrongly written code** when inheritance should be preferred over type checking:

```
switch (shape.Type)
{
   case Shape.Circle:
      shape.DrawCircle();
      break;
   case Shape.Square:
      shape.DrawSquare();
      break;
   …
}
```

It would make more sense if **Shape** was inherited by **Circle** and **Square**, which implement the virtual method **Shape.Draw()**.

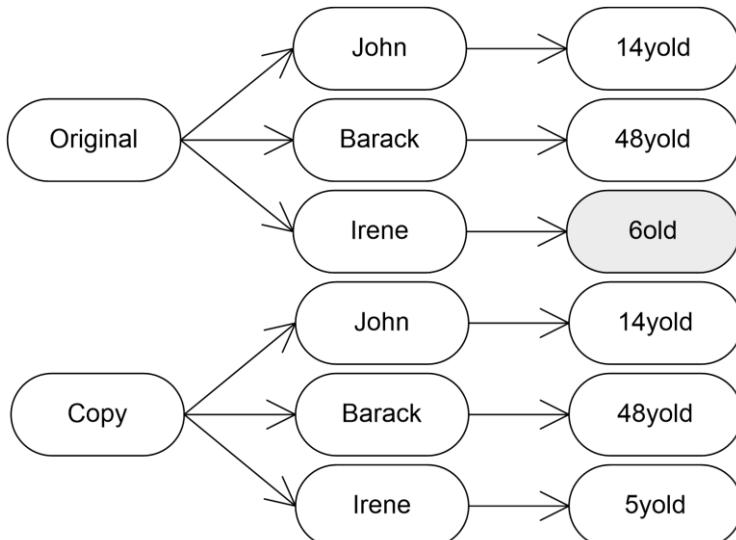
Encapsulation

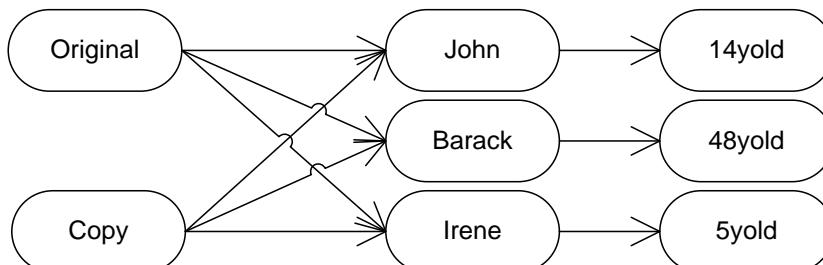
A good approach is to make all members **private**. Only those of them that should be visible from outside could be marked **protected**, or eventually **public**.

Implementation details should be hidden. The user of a high-quality class should not be aware of its inner-workings; he should only know what it does and how it is used.

Member-variables (fields) **should be hidden** behind properties. Public member-variables are a manifestation of low-quality code. Constants are an exception in this regard.

The public members of a class should be **consistent** with the abstraction represented by this class. Do not make assumptions about the usage scenario of a class.

Do not rely on undocumented, internal implementation logic.

Constructors

It is preferred that all class members are **initialized in the constructor**. Usage of an uninitialized class is dangerous. A half-initialized class is maybe even more dangerous. Initialize member-variables in the same order as they are declared.

Deep and Shallow Copy

When we assign values sometime we need to copy an object (make a duplicate). This can be done in two ways: **deep copy** or **shallow copy**.

Deep copies of an object are copies in which all member-variables are copied, and their member-variables also, and so on, until no other member-variables refer to objects. In a shallow copy, only the members at the first level are copied. **Example of deep copied object** and its members:

Shallow copies work differently. When a shallow copy is created, the original object and its copy **share some of their members**:

Shallow copies are dangerous because a change in one object leads to indirect changes in others. Notice how the change of Iren's age in the original does not affect the age of Iren in the copy when we use deep copies. With shallow copies, the change will be reflected in both places.

High-Quality Methods

The quality of our methods is of significant importance to creating high-quality software and its maintenance. They contribute to more readable and more comprehensible programs. Methods do help us **reduce the complexity** of our software, in order to make it more flexible and easier to modify.

It is up to us, to what extent we will benefit from these advantages. The higher the quality of our methods, the more we gain from their usage. In the next paragraphs we are introducing some of the basic principles for creating quality methods.

Why Should We Use Methods?

Before talking about good method names, let's spend some time and summarize the reasons for using methods.

A method solves a small problem. Many individual methods solve many small problems. Taken together, they solve a bigger problem – this is an illustration of the old Roman principle "Divide and conquer", which, in this case allows us to tackle smaller problems more easily.

With methods, the overall **complexity of a task is reduced**: complex problems are being split up into simpler ones, additional layers of abstraction are added, implementation details are hidden, and the risk of failure is lowered. **Code duplication is avoided** as well. Complex sequences of actions are hidden.

Since methods are the smallest reusable unit of code, their biggest advantage is the ability they give us to reuse code. In fact, that's exactly how methods emerged.

What Should a Method Do?

A method should **do the work described by its name, and nothing more**. If a method does not do what its name suggests, then either its name is wrong, or it does many things at the same time, or the method simply is incorrectly implemented. In any of these three cases, the method does not meet the requirements for code quality and should be refactored accordingly.

A method should either **do its expected job**, or should **inform for an error** and terminate. In .NET, informing for errors is done by **throwing an exception**. In case of invalid input, it is unacceptable for a method to return a wrong result. Instead, the method should inform the caller that it cannot do its job because the necessary preconditions are not met (such as invalid parameters being supplied, or an unexpected internal object state, etc.).

For example, suppose we have a method for reading the contents of a file. It should be called **ReadFileContents()** and should return **byte[]** or **string**, depending on whether we are treating the contents as binary or text. If the file does not exist or cannot be opened for whatever reason, the method should throw an exception rather than return an empty string or **null**.

Returning a neutral value (such as **null**) instead of an error message is generally **not recommended**, except in cases where that value does not collide with an error condition, such as a **Find()** method returning **null** because nothing was found. Otherwise, the caller loses its ability to handle the error, and the cause of the error is lost because of the lack of a richly informative exception.

A public method should either correctly accomplish exactly what its name suggests, or should inform the caller for an error by throwing an exception.

Any other behavior is incorrect!

The above rule has some exceptions when **private methods** are concerned. Unlike public methods, which should either work correctly or throw an exception, a compromise can be made for **private** methods. Since only the author of the class is supposed to call them, he should be aware of the validity of the passed arguments. Therefore, error conditions need not be handled because they can be predicted and prevented in the first place. But do not forget – this is still a compromise.

Two **examples of high-quality methods**:

```
long Sum(int[] elements)
{
   long sum = 0;
   foreach (int element in elements)
   {
      sum = sum + element;
   }
   return sum;
}

double CalcTriangleArea(double a, double b, double c)
{
   if (a <= 0 || b <= 0 || c <= 0)
   {
      throw new ArgumentException("Sides should be positive.");
   }
   double s = (a + b + c) / 2;
   double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
   return area;
```

```
}
```

Strong Cohesion and Loose Coupling

The rules regarding the logical relatedness of the responsibilities (**strong cohesion**) and the functional independence through a minimal amount of interaction with other methods and classes (**loose coupling**) are of a major importance when methods are concerned.

As we already explained, a method should **solve only one problem**, not many. A method should not solve numerous unrelated problems and should not have side effects. Otherwise, coming up with a precise and descriptive name is hard. This means that all of our methods should have **strong cohesion**, i.e. be concerned towards **solving a single problem**.

Methods should depend as little as possible on the rest of the methods in their class and on the methods / properties / fields in other classes. This concept is called **loose coupling**.

In the best-case scenario, a method should depend only on its parameters and not use any other data as its input or output. Such methods can be easily pulled out and **reused in another project**, because they are **unbound** to the environment in which they execute.

Sometimes methods depend on **`private`** variables declared within their class, or they alter the state of the object they belong to. This is not wrong and is entirely OK. In such a case we are talking about **coupling between the method and its class**. Such coupling is not problematic because the class and its internal data and logic are **encapsulated**: the whole class can still be moved into another project and reused without any modifications.

Most of the classes from .NET Common Type System (CTS) and .NET Framework define methods that depend only on the data within their class and the passed arguments. In standard libraries, the methods dependencies from external classes are minimal and that is why they are **easy to reuse**. The .NET Framework class library strongly follows the idea of **loose coupling**.

Whenever a method reads or modifies **global data** and depends on 10 additional objects, which must be initialized within the instance of its own class, it is considered a **coupled to its environment** and to all of these objects. This means that it functions in an overly complex way and is affected by too many external conditions, therefore the probability for an error is high. Methods that depend on too many external conditions are **hard to read**, **understand** and **maintain**. **Strong functional coupling is bad** and should be avoided as much as possible, because it often leads to **spaghetti code**.

Look at the same two methods like at our previous example. They are slightly modified and no longer fulfill the requirements of loose coupling and strong cohesion. **Do you spot errors?**

```
long Sum(int[] elements)
{
   long sum = 0;
   for (int i = 0; i < elements.Length; i++)
   {
      sum = sum + elements[i];
      elements[i] = 0; // Hidden side effect
   }
   return sum;
}

double CalcTriangleArea(double a, double b, double c)
{
   if (a <= 0 || b <= 0 || c <= 0)
   {
      return 0; // Incorrect result
   }
   double s = (a + b + c) / 2;
   double area = Math.Sqrt(s * (s - a) * (s - b) * (s - c));
   return area;
}
```

How Long Should a Method Be?

Throughout the years, research has been done regarding the optimal length of methods, but after all, a universal formula has not been found.

The practice shows that, in general, shorter methods (not longer than **a single screen**) should be preferred. Such methods are visible on the screen without scrolling and this simplifies their reading and understanding and the probability for making mistakes.

The longer a method, the more complex it becomes. Consequent modifications become considerably harder and more time-consuming than with shorter methods. These factors lead towards errors and harder maintenance.

The recommended length of a method is not more than **a single screen**, but this recommendation is only advisory. If a method fits on the screen, it is easier to read because scrolling is not needed. If a method is longer than one screen, we should think whether we can **split it up** into a few simpler methods. Since splitting is not always possible to be done in a meaningful way, the recommendation about method length is only advisory.

Although longer methods are not preferred, the latter should not be an absolute excuse for splitting up a method only to make it shorter. Methods should be **as long as necessary**.

Strong cohesion of methods is much more important than their length.

If we are implementing a complex algorithm and consequently come up with a longer, meaningful method, which does one thing and does it well, the length is not a problem.

In any case, we should at least **consider splitting up a longer method** into smaller methods solving particular subtasks, whenever the method becomes too long.

Method Parameters

One of the basic rules for ordering method parameters is that the primary one(s) should precede the rest. For example:

```
public void Archive(PersonData person, bool persistent) { … }
```

The opposite would be much more **confusing**:

```
public void Archive(bool persistent, PersonData person) { … }
```

Another rule is to have **meaningful parameter names**. A common **mistake** is to tie the parameter names to their type:

```
public void Archive(PersonData personData) { … }
```

Instead of the **meaningless personData** (which carries information only about the type), we can use a better name so that it becomes clear what kind of an object we are archiving:

```
public void Archive(PersonData loggedUser) { … }
```

If there are other methods with similar parameters, their **ordering should be consistent**:

```
public void Archive(PersonData person, bool persistent) { … }

public void Retrieve(PersonData person, bool persistent) { … }
```

It is important that no parameters are left **unused**. Unused parameters can only mislead the person who uses the code.

Parameters should not be used as local variables, that is, they should not be modified. Modifying method parameters makes the code harder to read and the logic becomes more convoluted. You can always **declare a new variable instead** of modifying a parameter. Conserving memory is not an excuse in such a scenario.

Implicit assumptions should be documented. An example would be to specify the measurement unit of a parameter to a method that computes the cosine of an angle – whether the angle is in radians or degrees, in case the name does not make it obvious.

The parameter count should not exceed 7. Seven is a special, **magic number**. It is proven in the psychology that the human mind cannot trace more than 7 (+/- 2) things simultaneously. As with parameter count, this recommendation is only advisory. Sometimes you need to pass more parameters. If that is the case, think about passing them as an object that represents a class with many fields. For example, instead of having an **AddStudent(…)** method with 15 parameters (name, address, contacts, etc.), you can reduce them by grouping logically related parameters into **separate objects**: **AddStudent(personalData, contacts, universityDetails)**. This way, each of the three parameters will contain a few fields inside, and the same information will be passed to the method, but in an easier to understand form.

Sometimes it is more appropriate, from a logical standpoint, to pass only one or a **few of the fields** of an object, rather than the **whole object**. This mostly depends on whether the method should be aware of the existence of this object or not. Suppose we have a method that calculates the final grade of a given student – **CalcFinalGrade(Student s)**. Because the final grade depends only on the previous grades and the rest of the student's data does not matter, it would be better if only the list of grades is passed – **CalcFinalGrade(IList<Grade>)**, instead of a **Student** object.

Proper Use of Variables

In this section we review a few **good practices for using local variables**.

Returning a Result

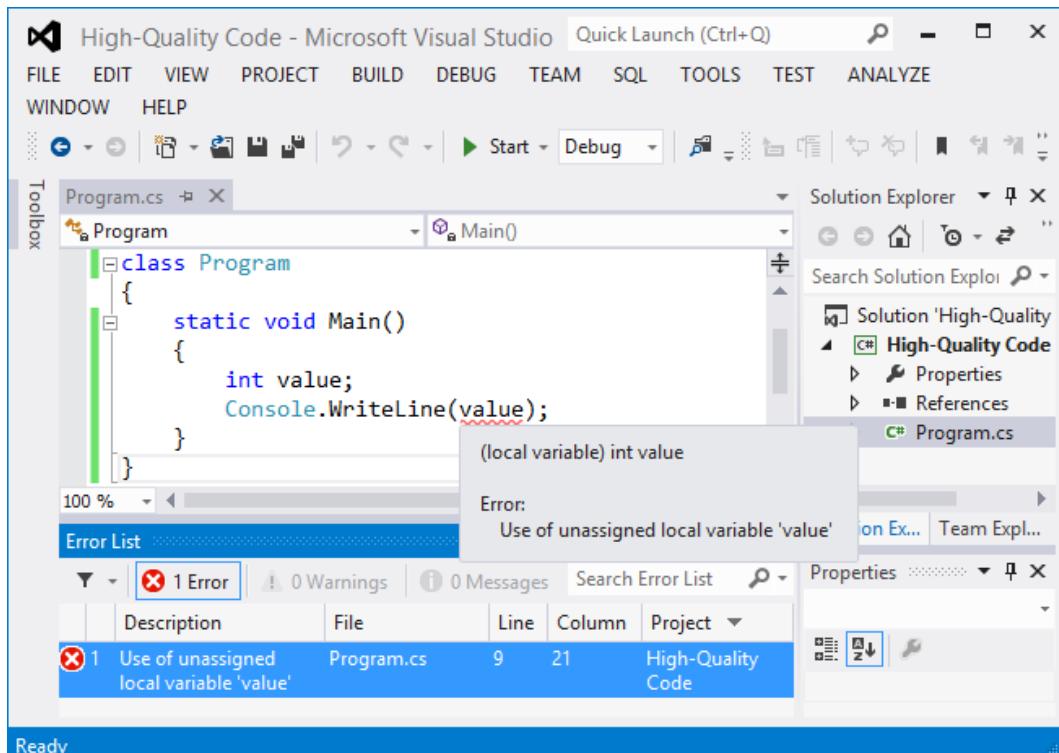
Whenever a result is returned, it should first be **saved in a variable**, before being returned. The following example does not hint at what exactly is returned:

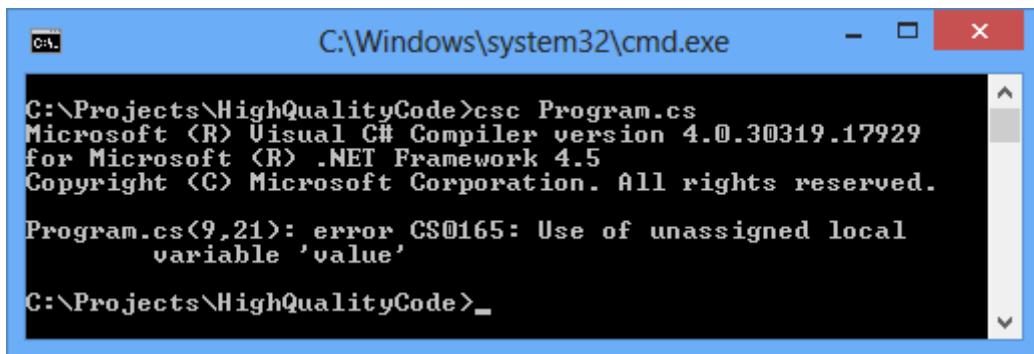
```
return days * hoursPerDay * ratePerHour;
```

It would be **better** like that:

```
int salary = days * hoursPerDay * ratePerHour;
return salary;
```

There are a few reasons for saving the result before returning it. For one, the additional variable contributes to **self-documenting** the code and makes it clear exactly what is returned. Another reason is tracing the returned value when **debugging** – we can stop the program from executing as soon as the

value is computed and then inspect it. A third reason is that it helps us avoid long expressions, which can become quite convoluted.

Principles for Initialization

In .NET, all **member-variables** (fields) belonging to a class are **initialized automatically** at the time of being declared (unlike C/C++). This is managed by the runtime and provides for a safer environment, less prone to errors caused from incorrectly initialized memory. All reference type variables are initialized to **null** and all primitive types to **0** (**false** for **bool**).

The compiler **forces the explicit initialization of all local variables**; otherwise a compile-time error is given. Here is an example that would cause such an error, because an attempt is made to use an uninitialized variable:

```
static void Main()
{
   int value;
   Console.WriteLine(value);
}
```

Here is how the compilation attempt looks like in Visual Studio:

Here is how the compilation attempt looks like in the console C# compiler:

C:\Windows\system32\cmd.exe

```
C:\Projects\HighQualityCode>csc Program.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

Program.cs(9,21): error CS0165: Use of unassigned local
        variable 'value'

C:\Projects\HighQualityCode>_
```

Let's look at the following more complex example:

```
int value;
if (condition1)
{
   if (condition2)
   {
      value = 1;
   }
}
else
{
   value = 2;
}
Console.WriteLine(value);
```

Fortunately, the compiler is smart enough to **analyze the control flow** and to catch such problems – the same error is thrown, because not all scenarios assign correctly the variable.

Note that adding an **else** to the nested **if** in the above code will make it compile without errors. If the variable is not initialized at its declaration, but is assigned to a value in all the possible paths of the control flow, the compiler will still be happy.

A good practice, however, is to initialize all variables explicitly at the time of their declaration:

```
int value = 0;
Student intern = null;
```

Partially Initialized Objects

Some objects, in order to be properly initialized, should have at least a few of their fields set. For example, an object of type **Person** should have valid values set for at least the **name** and the **family name** fields. This is something the compiler cannot prevent us from forgetting.

One way to solve this problem is to remove the default constructor (the one not taking any parameters) and to add one or more constructors, which take the sufficient data as their parameters, to enable the proper initialization of the object. This is the sole purpose of parameterized constructors.

Declaring a Variable within a Block or a Method

According to the coding convention for .NET, a local variable should be declared at the beginning of its enclosing block or method:

```
static int Archive()
{
  int result = 0; // beginning of method body
  // … Code …
}
```

Another example:

```
if (condition)
{
  int result = 0; // beginning of an "if" block
  // … Code …
}
```

An exception is made for variables declared within the initialization part of a **for**-loop:

```
for (int i = 0; i < data.Length; i++) { … }
```

The above recommendation is pretty disputable. Most good programmers prefer to declare a local variable **as close to its intended place of use** as possible. This helps reduce a variable's lifetime (refer to the next paragraph), and, at the same time, the probability for mistakes.

Scope, Lifetime and Span of Variables

The term **variable scope** actually denotes **how "famous" a variable is**. In .NET, three layers of variable scope exist: **static** variables, member-variables of a class (**fields**), and **local variables** inside a method.

Minimizing the Variable Scope

The **wider the scope** of a variable, the higher the probability that more code will be tied to it, thereby **increasing the level of coupling**. Since strong coupling is not desirable, variable scope should be as narrow as possible.

A good approach in using variables is to initially declare them with the **minimal scope**, and extend it only when necessary. This is a natural way of

assigning a variable the scope it needs. If you don't know what scope to use, start with **private** and if needed, switch to **protected** or **public**.

Static variables should best be **private** and accessing them should be controlled via appropriate methods.

Here is an **example of bad semantic coupling** based on a static variable, a horribly bad practice:

```
public class Globals
{
  public static int state = 0;
}

public class Genious
{
  public static void PrintSomething()
  {
    if (Globals.state == 0)
      Console.WriteLine("Hello.");
    else
      Console.WriteLine("Good bye.");
  }
}
```

If the **state** variable was marked **private**, such coupling would be impossible, at least not possible directly.

Minimizing the Variable Span

The **span** of a variable corresponds to the **average amount of lines between its occurrences in the code**. Considering minimal variable span, variables should be **declared and initialized as close as possible to their first occurrence** in the code, and not necessarily in the beginning of a method or a code block.

Keep the variable span as small as possible! This improves the code quality, readability, understandability and maintainability because less code needs to be inspected in order to read and understand the code.

Minimizing the Variable Lifetime

The **lifetime** of a local variable inside a method lasts between the **place of its declaration** (the beginning of a block, most usually), **until the end of the enclosing block**. Class fields (member-variables) exist as long as their class is instantiated. Static variables last throughout the entire execution of the program.

As you may guess, the **lifetime should be kept minimal**. This reduces the lines of code that you should consider at the same time when reading the

code. This will maximized the portion of the code you can safely ignore when you read the code. This will reduce the total complexity in your brain, because it works better with smaller and simpler pieces of code, right?

Minimizing the Variable Span and Lifetime – Example

Below we have an example of bad use of local variables (**unnecessarily large span**):

```
1  int count;
2  int[] numbers = new int[100];
3
4  for (int i = 0; i < numbers.Length; i++)
5  {
6     numbers[i] = i;
7  }
8  count = 0;
9
10 for (int i = 0; i < numbers.Length / 2; i++)
11 {
12    numbers[i] = numbers[i] * numbers[i];
13 }
14
15 for (int i = 0; i < numbers.Length; i++)
16 {
17    if (numbers[i] % 3 == 0)
18    {
19       count++;
20    }
21 }
22
23 Console.WriteLine(count);
```

lifetime = 23 lines

span = 23 / 4 = 5.75

In this example, the **count** variable's purpose is to count the numbers, which are evenly divisible by 3. It is used only in the last **for** loop, but is declared and initialized long before it.

What's wrong with the above code? If you try to read it and find how the **count** is calculated, you will need to inspect all its 23 lines, right? The code might be written differently and the variable count might be declared and zeroed just before the last **for**-loop. Thus if we need to read the code and find how **count** is calculated, we will need to inspect only 10 lines, not 23.

See below how the above code fragment can be refactored in order to **reduce the lifetime and span** of the **count** variable:

```
1  int[] numbers = new int[100];
```

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
	`for (int i = 0; i < numbers.Length; i++)`
	`{`
	`   numbers[i] = i;`
	`}`
	
	`for (int i = 0; i < numbers.Length / 2; i++)`
	`{`
	`   numbers[i] = numbers[i] * numbers[i];`
	`}`
	
	`int count = 0;`
	`for (int i = 0; i < numbers.Length; i++)`
	`{`
	`   if (numbers[i] % 3 == 0)`
	`   {`
	`      count++;`
	`   }`
	`}`
	
	`Console.WriteLine(count);`
		lifetime = 10 lines
		
		span = 10 / 3 = 3.33

It is important that the programmer tracks the usage of a particular variable, along with its scope, span and lifetime. The main objective is to **reduce the scope, the lifetime and the span** as much as possible. This leads to an important rule about correctly using variables:

Declare local variables as late as possible, immediately before using them for the first time. Initialize them at the time of declaration.

Variables with a **wider scope** and a longer lifetime should have **more descriptive names**, such as **totalStudentsCount** instead of **count**. That is because they occur at more locations within a larger piece of code, and hence the context around them is not going to be entirely clear.

Variables that span across just 4-5 lines can have **short and simple names**, for example **count**. They do not need long names because their purpose becomes clear from their limited context (a few lines), and ambiguities can rarely arise there.

Use of Variables – More Rules

A very important rule is to use a variable for one purpose only. The excuse that memory is conserved the other way is not generally convincing. If a variable is used for multiple different purposes, what name can we give it? Consider a variable that is used to count the number of students, and

occasionally the count of their grades. How would you call it: **count**, **studentsCount**, **gradesCount** or **studentsOrGradesCount**?

Use one variable for a single purpose only. Otherwise, an appropriate name cannot be found.

Unused local variables should not be present in the code. Their declarations alone are useless. Fortunately, most of the decent development environments do warn you about such anomalies.

The use of local variables with hidden meaning should be avoided. For example, John has left the variable **X** for Tom to see, so that he could get to the conclusion to implement another method that would use that same variable. Didn't get it? Good, let's hope you don't do it either.

Proper Use of Expressions

When using expressions, the simple rule is: **avoid complex expressions!** A complex expression is one that performs more than one thing:

```
for (int i = 0; i < xCoord.Length; i++)
{
  for (int j = 0; j < yCoord.Length; j++)
  {
    matrix[i][j] =
      matrix[xCoord[FindMax(i) + 1]][yCoord[FindMin(i) + 1]] *
      matrix[yCoord[FindMax(i) + 1]][xCoord[FindMin(i) + 1]];
  }
}
```

In the above sample we have a **complex calculation**, which fills a given matrix based on a computation over some coordinates. It is in fact very hard to understand what exactly is going on, because the used expressions are overly complex.

There are many reasons to avoid the use of complex expressions as in the above example. Let's mention a few:

- Code becomes **hard to read**. Therefore, tracing what is going on and whether the code is correct becomes hard, too.
- Code is **hard to maintain** – think about the effort involved in fixing an error, in case the code does not work as expected.
- Code is **hard to fix in case of defects**. If the above code throws **IndexOutOfRangeException**, how would we know exactly which array has been involved? It could be **xCoord** or **yCoord** or **matrix**, occurrences of which are all scattered within the expressions.

- Code is **hard to debug**. In case of an error, it would be much harder to debug a complex expression because it stays on a single line, and debuggers step through the code in terms of lines.

All of these reasons suggest that **writing complex expressions is harmful** and should be avoided. Instead of a single complex expression, we can write a few less complex ones and save them in variables with descriptive names. In this way the code becomes simpler, easier to read and understand and easier to maintain, debug and fix.

Let's **rewrite the above code** without using complex expressions:

```
for (int i = 0; i < xCoord.Length; i++)
{
   for (int j = 0; j < yCoord.Length; j++)
   {
      int maxStartIndex = FindMax(i) + 1;
      int minStartIndex = FindMax(i) - 1;
      int minXcoord = xCoord[minStartIndex];
      int maxXcoord = xCoord[maxStartIndex];
      int minYcoord = yCoord[minStartIndex];
      int maxYcoord = yCoord[maxStartIndex];
      matrix[i][j] =
         matrix[maxXcoord][minYcoord] *
         matrix[maxYcoord][minXcoord];
   }
}
```

Notice how simple and readable the code has become. Without knowing the exact calculation that this code is supposed to do, it is still hard to understand it, but at least we can debug it in case of an exception and find which line is causing it, and eventually fix it.

Do not write complex expressions. Only one operation should be performed at one line, otherwise the code becomes hard to read, maintain, debug and modify.

Use of Constants

Well written code should not contain "**magic numbers**" and "**magic strings**". Such constants are all the literals in a program having a value other than **0**, **1**, **-1**, **""** and **null** (with little exceptions).

In order to explain the concept why we need **named constants**, let's examine the code below. In this code we use the number **3.14159206** (**π**) three times (directly and in a formula), which introduces **duplicated code**. If, for example, we decide to increase the precision of this constant or change it, we will need to modify the program at three different locations:

```
public class GeometryUtils
{
  public static double CalcCircleArea(double radius)
  {
    double area = 3.14159206 * radius * radius;
    return area;
  }

  public static double CalcCirclePerimeter(double radius)
  {
    double perimeter = 6.28318412 * radius;
    return perimeter;
  }

  public static double CalcElipseArea(
    double axis1, double axis2)
  {
    double area = 3.14159206 * axis1 * axis2;
    return area;
  }
}
```

It comes to mind that it is better to define the repeating values only once on the code. In .NET such values are declared as **named constants** as follows:

```
public const double PI = 3.14159206;
```

After this declaration, the **PI** constant is accessible to the whole program and can be used an unlimited number of times. In case we need to change the value, we change it at one location only, and the changes are reflected everywhere. Here is how our **GeometryUtils** class looks after declaring the number **3.14159206** as a **named constant**:

```
public class GeometryUtils
{
  public const double PI = 3.14159206;

  public static double CalcCircleArea(double radius)
  {
    double area = PI * radius * radius;
    return area;
  }

  public static double CalcCirclePerimeter(double radius)
  {
```

```
        double perimeter = 2 * PI * radius;
        return perimeter;
    }

    public static double CalcElipseArea(
        double axis1, double axis2)
    {
        double area = PI * axis1 * axis2;
        return area;
    }
}
```

When to Use Constants?

The use of constants allows us to avoid the use of "magic numbers" and strings in our programs, and enables us to give names to the numbers and strings we use. In the above example not only we **avoided code duplication**, but we documented the fact that the number **3.14159206** is the well-known mathematical constant **π**.

Constants should be used whenever we need to use **numbers or strings whose origin and meaning are not obvious**. Constants should generally be defined for every number or string that is used more than once in a program (with some exceptions).

Here are a few typical cases in which **named constants** should be used:

- For **filenames** the program operates on. They need to be frequently changed and it is convenient to have them as named constants at the beginning of the program.
- For constants taking part in **mathematical expressions**. A good constant name improves the chance of understanding the formula.
- For **buffer sizes** and **sizes of memory blocks**. These sizes often need to be changed and that is why it is convenient to have them declared as named constants. Apart from that, using a constant named **READ_BUFFER_SIZE** rather than the number 8192 makes the code a lot more readable and comprehensible.

When Not to Use Constants?

Although many books recommend that all numbers and strings except **0**, **-1**, **1**, **""** and **null** are best declared as named constants, there are a few **exceptions in which declaring constants can be harmful**. Remember, declaring constants is made in order to improve the readability and the maintainability of the code. When a constant does not contribute to the readability of the code, you should avoid it.

Here are a few situations in which using a named constant can be harmful:

- **Error messages** and other messages intended for the user ("Enter your name", for example). Making such strings named constants will actually hinder the readability.
- **SQL queries** in named constants are not recommended (in case you are using a database, queries are usually written in SQL, and that is usually a string in the terms of the programming language).
- Button labels, dialog box titles, menu entries and captions of other **UI components** should not be declared as named constants.

The .NET Framework provides libraries that facilitate internationalization and allow exporting all the messages, captions and labels from the UI in special **resource files**. These are not constants, however. This approach is encouraged if the program you are writing will have to be internationalized.

Use named constants to avoid the usage and duplication of magic numbers and strings, and mostly to improve code readability. If the introduction of a named constant hinders the readability, better leave the hardcoded value in the code!

Proper Use of Control Flow Statements

Control flow statements are represented by loops and conditional statements. We are going to review the **good practices** for using them properly.

With or Without Curly Brackets?

Loops and conditional statements allow their body to not be surrounded by brackets, in case the body consists only of a single statement. This can be **dangerous**. Consider the following example:

```
static void Main()
{
  int two = 2;
  if (two == 1)
    Console.WriteLine("This is the ...");
    Console.WriteLine("... number one.");
  Console.WriteLine(
    "Example of an if-clause without curly brackets.");
}
```

We are expecting to see only the last sentence, aren't we? The result is a bit unexpected:

```
... number one.
Example of an if-clause without curly brackets.
```

That is because an **if**-statement without curly brackets only takes the first statement as its body, regardless of the indentation, which makes matters confusing.

	Always enclose the body of loops and conditional statements in curly brackets – { and }.

Proper Usage of Conditional Statements

Conditional statements in C# are represented by the **if-else** and the **switch-case** statements.

```
if (condition)
{

}
else
{

}
```

Deep Nesting of Conditional Statements

Deep nesting of if statements is a bad practice because it obstructs the comprehensibility of the code:

```
1   private int Max(int a, int b, int c, int d)
2   {
3      if (a < b)
4      {
5         if (b < c)
6         {
7            if (c < d)
8            {
9               return d;
10           }
11           else
12           {
13              return c;
14           }
15        }
16        else if (b > d)
17        {
18           return b;
19        }
```

```
20        else
21        {
22          return d;
23        }
24      }
25      else if (a < c)
26      {
27        if (c < d)
28        {
29          return d;
30        }
31        else
32        {
33          return c;
34        }
35      }
36      else if (a > d)
37      {
38        return a;
39      }
40      else
41      {
42        return d;
43      }
44    }
```

This code is hardly readable because of the deep nesting. In order to improve it, we could **introduce a few more methods** where parts of the logic are exported and isolated. Here is how we could do that:

```
1   private int Max(int a, int b)
2   {
3     if (a < b)
4     {
5       return b;
6     }
7     else
8     {
9       return a;
10    }
11  }
12
13  private int Max(int a, int b, int c)
14  {
15    if (a < b)
```

```
16     {
17         return Max(b, c);
18     }
19     else
20     {
21         return Max(a, c);
22     }
23 }
24 
25 private int Max(int a, int b, int c, int d)
26 {
27     if (a < b)
28     {
29         return Max(b, c, d);
30     }
31     else
32     {
33         return Max(a, c, d);
34     }
35 }
```

Extracting parts of the code into **separate methods** is the easiest and most efficient way to reduce the level of nesting of a group of conditional statements, while preserving their logic.

The **refactored method** is split into a few smaller ones. The overall length of the code has been decreased by 9 lines. Each of the new methods is simpler and easier to read. As a side benefit, we get two methods that can be easily reused for other purposes.

Proper Use of Loops

Proper use of the different looping constructs is very important to the creation of quality software. In the next paragraphs we outline some of the principles, which help us decide when, and how to use a particular loop construct.

Choosing an Appropriate Looping Construct

If we are not able to decide whether to use **for**, **while** or **do-while** loop, we can easily pick up one, adhering to the following principles.

If we need a loop that will execute a **fixed number of times**, a **for**-loop is a good fit. This kind of loop is used in the most basic situations where interrupting the control is not necessary. The initialization, the check of the condition and the incrementing are all in the **for**-construct and the loop body does not care about that. The value of the counter should not be altered within the body.

If it is necessary to check **some conditions in order to stop** the execution of the loop, then it is probably better to pick a **while** loop. A **while** loop is suitable in cases where the exact number of iterations is not known. The execution there continues until the exit condition has been encountered. If the prerequisites for using a **while** loop are in place, but the loop body must unconditionally **execute at least once**, a **do-while** loop should be used instead.

Do Not Nest Too Many Loops

As with conditional statements, **deep nesting of loops is a bad practice**. Deep nesting usually happens because of a large number of loops and conditional statements residing in one another. This makes the code hard to read and maintain. Such code can easily be improved by moving away parts of it into separate methods. Modern development environments can do such refactoring automatically (we talk about that in the code refactoring section).

Defensive Programming

Defensive programming is a term denoting a practice towards **defending the code from incorrect data**. Defensive programming keeps the code from errors that nobody expects. It is implemented by **checking the validity of all input data**. This is the data coming from external sources, input parameters of methods, configuration files and settings, input from the user, and even the data from another local method.

The main idea behind defensive programming is that methods should **check their input parameters** (and other input data) and inform the caller when the object's internal state or the input parameters are incorrect.

Defensive programming requires that **all data is checked**, even if it is coming from a trusted source. If this trusted source happens to have a bug, the bug will be found earlier and more easily.

Defensive programming is implemented through **assertions**, **exceptions** and other means of error handling.

Assertions

Assertions are **special conditions** that should **always be met**. If not met, they throw an error message and the program terminates.

A quick **example of assertion** in C# is shown below:

```
void LoadTemplates(string fileName)
{
  bool templatesFileExist = File.Exists(fileName);
  Debug.Assert(templatesFileExist,
    "Can't load templates file: " + fileName);
}
```

Assertions vs. Exceptions

Exceptions are announcements for an error or for an unexpected event. They inform the programmer using the code for an error. Exceptions can be caught and program execution can still continue.

Assertions produce fatal errors. They cannot be caught or handled, because they are meant to indicate a bug in the code. A failed assertion causes the program to terminate.

Assertions can be turned off. The concept is to have them turned on only at the time of developing, in order to find as many bugs as possible. When turned off, the conditions are no longer checked. Turning off the assertions is plausible when the software goes to production, since these checks are affecting the performance and the messages are not always meaningful to the end user.

If a particular check should continue to exist when the software goes to production (for example, checking the input that comes from the user), it should not be implemented as an assertion in the first place. Exceptions should be used in such cases instead.

Assertions should only be used for conditions that, if not met, it is due to a bug in the program.

Defensive Programming with Exceptions

Exceptions provide a powerful mechanism for **centralized handling of errors and unusual conditions**. They are covered in details in the "Exception Handling" chapter.

Exceptions allow problematic situations to be handled at many levels. They ease the writing and the maintenance of reliable program code.

Another difference between exceptions and assertions is that, in defensive programming, exceptions are mainly used for protecting the public interface of a class or component. This provides for a **fail-safe mechanism**.

If the **Archive** method described above was a part of the public interface of an archiving component rather than an internal method, it would have to be implemented as follows:

```
public int Archive(PersonData user, bool persistent)
{
  if (user == null)
  {
    throw new StorageException("null parameter");
  }

  // Do some processing
```

```
    int resultFromProcessing = …

    Debug.Assert(resultFromProcessing >= 0,
      "resultFromProcessing is negative. There is a bug!");

    return resultFromProcessing;
}
```

The **Assert** still remains because it is validating a variable created within the method itself.

Exceptions should be used to inform other parts of the code for problems that should not be ignored. **Throwing an exception** is reasonable only in situations when an **abnormal condition has occurred**. For more information on the situations considered exceptional, refer to the "Exception Handling" chapter.

If a particular problem can be **handled locally**, the handling should be performed in the method itself and no exceptions should be thrown. If a problem cannot be handled locally, the exception should be thrown to the caller.

The thrown exceptions should be at an appropriate level of abstraction. For example **GetEmployeeInfo()** could throw **EmployeeException**, but not **FileNotFoundException**. The last example throws **StorageException** rather than **NullReferenceException**.

Code Documentation

The C# specification allows putting comments in the code. We are already familiar with the basic principles for **writing comments**. In the next few paragraphs we explain how to write **effective comments**.

Self-Documenting Code

A very important point to remember is that comments in the code are not the primary source of documentation. **Good programming style provides the best documentation**. Self-documenting code rarely needs comments because its intention becomes clear directly by reading it. Self-documenting code means a code that is **easy-to-read** and **easy-to-understand** without having comments inside.

The best way to document the code is to write quality code. Bad code should not be documented but should rather be rewritten! Comments are only a complement to the well-written code.

Properties of Self-Documenting Code

Self-documenting code boasts a good structure: everything mentioned in this chapter matters. The implementation should be as simple as possible so that anyone can understand it.

Self-Documenting Code – Important Questions

In order to qualify our code as self-documenting, there are a few questions we should ask ourselves:

- Is the **class name** appropriate and does it describe its main purpose?
- Is the **public interface** of the class intuitive to use?
- Does the **name of a method** describe its main purpose?
- Is every method performing a **single, well-defined task**?
- Are the **names of the variables** corresponding to the intent of their use?
- Are loops performing only a **single task**?
- Are conditional statements **deeply nested**?
- Does the organization of the code illustrate its **logical structure**?
- Is the **design clear** and unambiguous?
- Are **implementation details hidden** as much as possible?

Effective Comments

Comments can sometimes do **more harm than good**. Good comments **do not repeat the code** and do not explain it line by line: they rather clarify its idea. Comments should describe at a higher level what our intentions are. Comments enable us to think better about what we want to implement.

Here is an **example of bad comments**, which, instead of making the code more comprehensible, are actually annoying:

```
public List<int> FindPrimes(int start, int end)
{
  // Create new list of integers
  List<int> primesList = new List<int>();
  // Perform a loop from start to end
  for (int num = start; num <= end; num++)
  {
    // Declare boolean variable, initially true
    bool prime = true;
    // Perform loop from 2 to sqrt(num)
    for (int div = 2; div <= Math.Sqrt(num); div++)
```

```
    {
      // Check if div divides num with no remainder
      if (num % div == 0)
      {
        // We found a divider -> the number is not prime
        prime = false;
        // Exit from the loop
        break;
      }
      // Continue with the next loop value
    }

    // Check if the number is prime
    if (prime)
    {
      // Add the number to the list of primes
      primesList.Add(num);
    }
  }

  // Return the list of primes
  return primesList;
}
```

If, instead of writing naive comments, we write comments to **clarify the unobvious facts in the code**, comments can be very useful. Here is how the same code can be commented meaningfully:

```
/// <summary>Finds the primes from a range [start, end] and
/// returns them in a list.</summary>
/// <param name="start">Top of range</param>
/// <param name="end">End of range</param>
/// <returns>a list of all the found primes</returns>
public List<int> FindPrimes(int start, int end)
{
  List<int> primesList = new List<int>();
  for (int num = start; num <= end; num++)
  {
    bool isPrime = IsPrime(num);
    if (isPrime)
    {
      primesList.Add(num);
    }
  }
  return primesList;
```

```
}

/// <summary>Checks if a number is prime by checking for any
/// dividers in the range [2, sqrt(number)].</summary>
/// <param name="number">The number to be checked</param>
/// <returns>True if prime</returns>
public bool IsPrime(int number)
{
   for (int div = 2; div <= Math.Sqrt(number); div++)
   {
      if (number % div == 0)
      {
         return false;
      }
   }

   return true;
}
```

The logic of the code is obvious and **does not need any comments**. In such case it is sufficient only to describe what are the particular method's purpose and its general idea, in a single sentence.

In order to write effective comments, it is desirable to use pseudo-code, whenever possible. Comments should be written **at the time the code is written**, not after that.

Productivity (i.e. writing code quickly) is never a good excuse for not writing comments. Everything that is not instantly obvious should be documented. Writing too much unnecessary comments is as bad as not having any at all.

Bad code cannot be improved by putting more comments. It should instead be rewritten or refactored.

XML Documentation in C#

You might have already noted the special comments in the code that explain the purpose of a class or a method and its parameters:

```
/// <summary>Finds the primes from a range [start, end] and
/// returns them in a list.</summary>
/// <param name="start">Top of range</param>
/// <param name="end">End of range</param>
/// <returns>a list of all the found primes</returns>
public List<int> FindPrimes(int start, int end)
{ … }
```

This special style of documentation built-in the C# source code is called **XML documentation**. It is enclosed in the triple comments **///** and uses few special XML tags: to document a type / method summary (**<summary>**), to describe method's parameters (**<param name="…">**), to describe a method's return value (**<returns>**), to document exceptions that eventually might be thrown (**<exception cref="…"**), to make a cross-reference link to related type (**<seealso cref="…"/>**), to describe some remarks (**<remarks>**), to give an example how to use the type / method (**<example>**), etc.

Using XML-style documentation in the source code has several **advantages**:

- The XML documentation is **built-in** the source code itself.
- The XML documentation is **automatically processed by Visual Studio** and is displayed in its autocomplete feature.
- The XML documentation can be **compiled into an MSDN-style web site or e-book** (in CHM format) through specialized tools like Sandcastle (http://shfb.codeplex.com).

More about writing and using XML documentation can be found in MSDN Library: http://msdn.microsoft.com/en-us/library/b2s063f7.aspx.

Code Refactoring

The term "**refactoring**" appears in 1993 and is popularized by **Martin Fowler** in his book with the same name. This book reviews a lot of techniques for code refactoring (called **refactoring patterns**). We are going to mention a few of them.

A program needs refactoring in case of **code duplication**. Code duplication is dangerous because a change in one place requires that all the other duplicated code be changed as well. The latter is error-prone and inconsistencies can arise therefore. Avoiding code duplication can be achieved by putting the particular piece of code in a method, or by moving common functionality to base classes.

Refactoring is necessary for methods, which have grown over time. The **excessive length** of a method is a good reason to think about splitting it up logically into few smaller and simpler methods.

Deeply nested constructs are another reason for refactoring. They can be eliminated by taking out a block of code into a method.

Classes that do not provide a sufficiently good **level of abstraction** or ones that perform **unrelated tasks** (weak cohesion) are candidates for refactoring as well.

Long parameter lists and **public fields** should also go to the fix-it list. **Tightly coupled classes** go in the same category.

Refactoring at Data Level

A good practice is to **avoid magic numbers** scattered throughout the code. They should be replaced by named constants. Variables with unclear names should be **renamed**. Long conditional expressions can be refactored into **separate methods**. Variables can be used to hold the intermediate results of expressions. A group of data that always appears together can be refactored into a separate class. Related constants should be grouped into enumerations.

Refactoring at Method and Class Level

Within a longer method, all tasks that are unrelated to its main purpose are better **moved into separate methods**. Similar tasks should be grouped in common classes, similar classes – in a common package. If a group of classes have common functionality, it should be moved into a base class.

Circular dependencies between the classes should not exist, they should be removed. In most cases the more common class has a reference to the more specialized class (parent-child relationship).

Unit Testing

Unit testing means to **write a program that tests a certain method or class**. A typical unit test executes the method that should be tested, passes a sample data to (parameters and object states) and checks whether the method's result is correct (for this sample data), i.e. whether the method does exactly what it should do and whether it does it correctly.

A single method usually is tested by **several unit tests**, each implementing a different testing scenario. First, the **typical case** is checked. Then the **border cases** are checked. The border cases are special cases which could need special processing logic, e.g. the largest or the smallest possible value, the first or last element, etc. Finally the method is tested with **incorrect data** and an exception is expected to be thrown. Sometimes a **performance test** may be involved to check whether the method is fast enough.

Unit Testing – Example

Let's see a small example – a method that **sums an array of numbers**:

```
static int Sum(int[] numbers)
{
    int sum = numbers[0];
    for (int i = 1; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
    return sum;
}
```

The above method may look correct but in fact **it has several bugs** that we will catch through unit testing. Let's first test the **typical case**:

```
if (Sum(new int[] {1, 2}) != 3)
  throw new Exception("1 + 2 != 3");
```

Seems like the **Sum(…)** method is working correctly in its typical case: the sum 1+2 is 3 (as expected) and the above code produces nothing. The above piece of code is called "**unit test**". It tests a certain method, class or other functionality against certain testing scenario and notifies us if the code behaves unexpectedly. If the test passes, the code produces no result.

Let's now test the **border cases**. What will happen if we **sum only one number**? Let's try:

```
if (Sum(new int[] {1}) != 1)
  throw new Exception("Sum of 1 != 1");
```

Seems like our method **still works correctly**. New let's try to sum an empty list of numbers. Their sum should be 0, right? Let's try this:

```
if (Sum(new int[] {}) != 0)
  throw new Exception("Sum of 0 numbers != 0");
```

The above code produces unexpected exception in the **Sum(…)** method:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was
outside the bounds of the array.
```

We **found a bug**, right? Let's fix it. We could start summing from 0 instead from the first element in the array (which could be missing when an empty array is passed as an argument). Below is the fixed code:

```
static int Sum(int[] numbers)
{
  int sum = 0;
  for (int i = 0; i < numbers.Length; i++)
  {
    sum += numbers[i];
  }
  return sum;
}
```

We repeat our last test (summing an empty array of numbers) and it now **passes correctly**. Next we could try other **special (border) cases**, e.g. summing negative numbers:

```
if (Sum(new int[] {-1, -2}) != -3)
  throw new Exception("-1 + -2 != -3");
```

What else to try? Seems like our method work correctly. We could try to find some **extreme case** when the method eventually fails. What will happen if we sum too big numbers? **Int32** cannot hold **too big integers**. Let's try:

```
if (Sum(new int[] { 2000000000, 2000000000 }) != 4000000000)
  throw new Exception("2000000000 + 2000000000 != 4000000000");
```

We **found another bug** in our method for summing numbers:

```
Unhandled Exception: System.Exception: 2000000000 +
2000000000 != 4000000000
```

Obviously the integer type **Int32** overflows and this causes incorrect result when summing too large numbers. Let's fix this. We can use **long** to keep the sum of the numbers instead of **int**:

```
static long Sum(int[] numbers)
{
  long sum = 0;
  for (int i = 0; i < numbers.Length; i++)
  {
    sum += numbers[i];
  }
  return sum;
}
```

Let's repeat the last test. Now it works. What else to test? What will happen if we pass **null** as an argument to the **Sum(…)** method? The recommendations about high-quality methods say that "**a method should return what its name says or throw and exception if it cannot do its job**". So our method should throw an exception if we try to sum a **null** array. We could test this in the following way:

```
try
{
  Sum(null);
  // An exception is expected --> the test fails
  throw new Exception("Null array cannot be summed.");
}
catch (NullReferenceException)
{
  // NullReferenceException is expected --> the test passes
}
```

The above unit test is a bit more complicated: it **expects an exception** and if it is not thrown, it fails.

What else to test? Maybe we could make a **performance test**? For example we could sum 10,000,000 numbers and expect this will take time less than 1 second (we assume a modern computer will run the tests):

```
DateTime startTime = DateTime.Now;
int[] arr = new int[10000000];
for (int i = 0; i < arr.Length; i++)
{
  arr[i] = 5;
}
if (Sum(arr) != 50000000)
  throw new Exception("5 + ... (10000000 times) != 50000000");
DateTime endTime = DateTime.Now;
if (endTime - startTime > new TimeSpan(0, 0, 1))
  throw new Exception("Performance issue: summing 10000000 " +
  "numbers takes more that 1 second");
```

The performance test passes without any issues.

We repeat all tests again to ensure that after the modifications we made all tests are still working correctly. All tests pass! We can now be confident that the **Sum(…)** method works correctly (even in unusual situations) and it is **well tested**. Let's think about what are the **benefits** if we test in similar manner all methods in our code.

Benefits of Unit Testing

Unit testing has many benefits for our code quality. Let's discuss the most important of them:

- Unit testing **significantly improves the code quality**. If the unit tests are well written and the entire functionality is covered, the code is expected to be **bug free**. In practice it is very hard to cover with tests any possible scenario so unit testing only dramatically reduces the number of bugs but does not make the code bug free.

- Unit testing allows the tests to be **executed many times**, continuously, e.g. at every hour. If some test fails, the problem is caught almost instantly. In software engineering the practice of executing the unit tests continuously is called "**continuous integration**".

- The code quality is preserved every time the method is modified. This **dramatically simplifies the maintenance**. If we change the algorithm inside some method or class and we have covered it well with tests, we will be sure that the new algorithm behaves the same way like the old.

- Unit tests **allow code refactoring** without worrying of something being broken. It can happen that we refactor the code to improve its internal quality but by mistake after the refactoring the code does not work correctly in all special cases.

All serious software development companies and software products use unit testing. For example if you download the source code of Firefox, you will notice that half of the code is written to perform unit tests over the other half of code. In practice it is **impossible to write complex product** (like for example MS Word or Android OS or Firefox browser) **without unit testing**.

Benefits of Unit Testing – Example

Let's see one of the benefits of unit testing: the ability to change the internal implementation of a method and re-test it to ensure the new implementation works as expected. Consider the following new implementation of the **Sum(…)** method that uses the **Sum()** extension method from **System.Linq**:

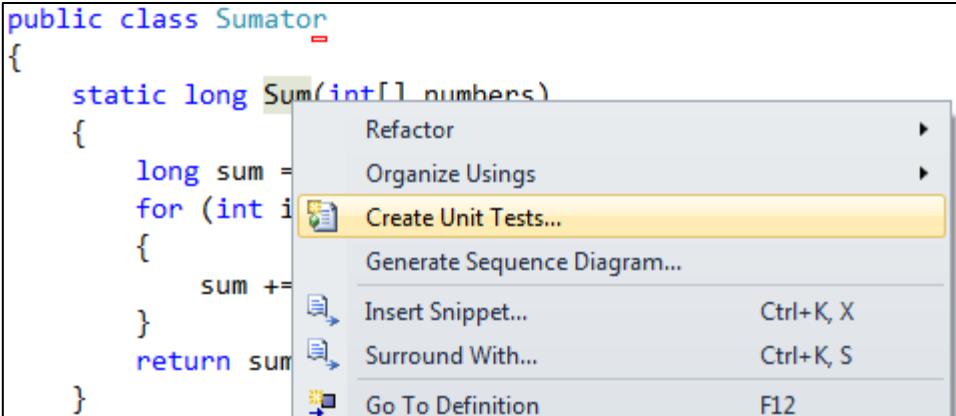
```
using System;
using System.Linq;

static long Sum(int[] numbers)
{
  return numbers.Sum();
}
```

We will explain how the above code works in the next chapter "Lambda Expressions and LINQ". Now let's test it to ensure this code behaves as expected. If we run the same set of tests we discussed above, we will find a problem: **two of our tests do not work**. The first failing test is:

```
if (Sum(new int[] { 2000000000, 2000000000 }) != 4000000000)
  throw new Exception("2000000000 + 2000000000 != 4000000000");
```

We found a bug in our new implementation of our **Sum(…)** method: instead of returning the correct result it produces **System.OverflowException**. We cannot find an easy solution to this problem so we can either assume that summing too big numbers will not be supported and modify the test to expect **OverflowException** or we can rewrite the **Sum(…)** method with a new implementation.

If we pass ahead, we will find that one more unit test fails: when we try to sum a **null** array, we will get **System.ArgumentNullException** instead of **NullReferenceException**. This is easy to fix by modifying the unit test code:

```
try
{
  Sum(null);
  // An exception is expected --> the test fails
  throw new Exception("Null array cannot be summed.");
}
catch (ArgumentNullException)
{
  // NullReferenceException is expected --> the test passes
}
```

Now all unit tests works correctly. The conclusion form the above experience is that when we modify the code and **a unit test fails, either the tested code is incorrect, or the unit test is incorrect**. In both cases we are notified that our new code behaves differently than our old code. This is very important in software engineering process. When we develop a complex software product, we want the features that work in its current version to continue to works the same way in all its next versions. For example, if we work on MS Word and we add PDF export for its next version, we want to be sure that saving in DOCX format still works after the PDF export is introduced.

Unit Testing Frameworks and Tools

To simplify writing unit tests and execute them many **unit testing frameworks** and tools have emerged. In C# we can use **Visual Studio Team Test (VSTT)** or **NUnit** frameworks to simplify the process of writing tests, asserting test conditions and executing test cases and test suites.

Unit Testing with Visual Studio Team Test (VSTT)

If you have installed Visual Studio 2010 edition which supports unit testing (e.g. Visual Studio 2010 Ultimate), you will have the **[Create Unit Tests …]** feature in the popup menu when you right click at some method in your C# code:

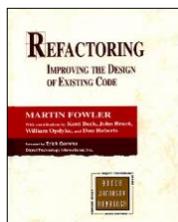
```
public class Sumator
{
    static long Sum(int[] numbers)
    {
        long sum =
        for (int i
        {
            sum +=
        }
        return sum
    }
```

Refactor
Organize Usings
Create Unit Tests...
Generate Sequence Diagram...
Insert Snippet... Ctrl+K, X
Surround With... Ctrl+K, S
Go To Definition F12

The above feature was introduced in VS 2010 and **is missing in VS 2012** for unknown reason. So if you are using Visual Studio 2012, you need to create a unit test project by hand (File → New Project → Unit Test Project).

The **unit tests** in Visual Studio Team Test look like the following:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class SumatorTest
{
  [TestMethod]
  public void SumTestTypicalCase()
  {
    int[] numbers = new int[] { 1, 2 };
    long expected = 3;
    long actual = Sumator_Accessor.Sum(numbers);
    Assert.AreEqual(expected, actual);
  }

  [TestMethod]
  public void SumTestOverflow()
  {
    int[] numbers = new int[] { 2000000000, 2000000000 };
    long expected = 4000000000;
    long actual = Sumator_Accessor.Sum(numbers);
    Assert.AreEqual(expected, actual);
  }

  [TestMethod]
  [ExpectedException(typeof(NullReferenceException))]
  public void SumTestNullArray()
  {
    Sumator_Accessor.Sum(null);
  }
}
```

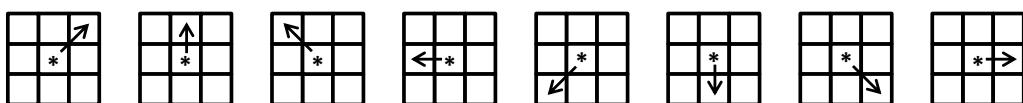
A detailed explanation of VSTT will not be given in this book, but anyone could research how to use unit testing in Visual Studio. As you see from the example above, VSTT simplifies unit testing by introducing **test classes** and **test methods**. Each test method has a meaningful name which and tests a certain test case. VSTT can test private methods, can set time limit for the test execution and can expect exception to be thrown by certain test case – things that **simplify writing the testing code**. Visual Studio can execute and visualize the results of the test execution:

Test run failed Results: 1/3 passed; Item(s) checked: 2

		Result	Test Name	Project	Error Message
☐		Passed	SumTestTypicalCase	TestSumator	
☑		Failed	SumTestNullArray	TestSumator	Test method SumatorTest.SumT
☑		Failed	SumTestOverflow	TestSumator	Test method SumatorTest.SumT

Additional Resources

We hope this chapter made the first steps in making you a real high-quality software engineer. If you want to learn more about writing quality code, you might refer to these additional resources:

The Bible of quality programming code is called "**Code Complete**" and its second edition was published in 2004. Its author, **Steve McConnell**, is a world-famous expert on writing quality software, a former Microsoft employee. The book contains a lot more examples and more general practices for writing high-quality code.

Another good book on software quality is **Martin Fowler**'s "**Refactoring: Improving the Design of Existing Code**". This book is considered to be the Bible of code refactoring. Terms such as "extract method", "encapsulate field", "extract constant" and other basic modern refactoring patterns were first described in this book.

The free training course "**High-Quality Code**" @ **Telerik Software Academy** – http://codecourse.telerik.com. It provides comprehensive teaching materials, presentations, examples, homework assignments and videos (in Bulgarian) about writing high-quality code and high-quality software, unit testing and code refactoring.

Exercises

1. Take the code from the first example in this chapter and **refactor it** to meet the quality standards discussed in this chapter.

2. **Review your own code** from the exercises from the previous chapters and find the mistakes you have made. **Refactor the code** to improve its quality. Think how you can avoid such mistakes and bad coding style in the future.

3. Open **other people's code** and **try to understand it** only by reading the code itself. Is everything obvious at first sight? What would you change in that code, how would you write it?

4. Review the classes from **.NET Common Type System** (CTS). Can you find examples of **low-quality code**?

5. Have you used or seen any **coding conventions**? Having read this chapter, would you consider them good or bad?

6. We are given a square matrix of **n** x **n** cells. A **rotating walk in the matrix** is walk that starts from the top left corner of the matrix and goes in down-right direction. When no continuation is available at the current direction (either the matrix wall or non-empty cell is reached), the **direction is changed** to the next possible direction clockwise. The eight possible directions are as follows:

When no empty cell is available at all directions, **the walk is restarted** from an empty cell at the smallest possible row and as close as possible to the start of this row. When no empty cell is left in the matrix, **the walk is finished**. Your task is to write a program that reads from the console an integer number **n** (1 ≤ **n** ≤ 100) and displays the filled matrix on the console.

Sample input:

```
n = 6
```

Sample output:

```
 1  16  17  18  19  20
15   2  27  28  29  21
14  31   3  26  30  22
13  36  32   4  25  23
12  35  34  33   5  24
11  10   9   8   7   6
```

Download a sample **low-quality solution** of that problem from here: http://introcsharpbook.googlecode.com/files/High-Quality-Code.rar.

Refactor the code so that it meets the recommended standards for quality code stated in this chapter. Note that fixing bugs in the solution might be necessary if it does not work correctly.

Solutions and Guidelines

1. Use [Ctrl+K, Ctrl+F] in Visual Studio to **reformat the code** and see the differences. Then **rename** the variables, omit the unnecessary statements and variables, and make the output that is printed more meaningful.

2. Pay special attention to the **recommendations for quality code** from this chapter. Remember your most frequent mistakes and try to avoid them. The most often problem with the code written by inexperienced programmers is the **naming**. You can use the “**rename**” feature in Visual Studio (shortcut [Ctrl+R, Ctrl+R]) to rename the identifiers in the code

when necessary. You may need to reformat your code through [Ctrl+K, Ctrl+F] in Visual Studio. You may need to **extract pieces of code in separate method**. This can be done through "Refactor" → "**Extract Method …**" feature in Visual Studio (shortcut [Ctrl+R, Ctrl+M]).

3. Take some **well-written software as an example** (e.g. Wintellect Power Collections for .NET – http://powercollections.codeplex.com). You would probably find things that you would write in a different way, or things that this chapter suggests **should be done differently**. Deviations are possible and are completely normal. One of the biggest differences between low-quality and high-quality code is the **consistency in following the rules**. The rules in different projects may be different (e.g. different formatting style, different documentation style, different naming style, different project structure, etc.) but the **general recommendations for writing high-quality code** will be followed.

   Take another example: **bad code** that is hard to read, understand and maintain. You may find many examples in Internet but to save time you may look at the projects from the "High-Quality Code" course at Telerik Software Academy (May 2011): https://qualitycode.googlecode.com/svn/trunk/2011/Exams/Final-Projects-19-May-2011/High-Quality-Code-2011-Final-Projects.rar. There are C#, Java, C++ and PHP projects with low-quality code that needs deep refactoring and quality improvement.

4. The code from CTS is written by engineers with an extensive experience and you can **rarely encounter low-quality code** there. Despite of that, anomalies such as using complex expressions and inappropriately named variables can still be seen. Try to find some examples of bad coding practices in CTS. Use **JustDecompile** or other decompilation tool because the source code of CTS is unavailable. Keep in mind that local variable names and comments in the code are lost when the code is compiled and decompiled so the variable names might be incorrect.

   Instead of decompiling the .NET CTS you may look at the **source code of Mono** (the open-source .NET implementation for Linux) at GitHub: https://github.com/mono/mono/tree/master/mcs/class/corlib. An example of code that needs improvement is the `Dictionary<K,T>` implementation in Mono: Dictionary.cs.

5. Just answer based on your personal experience. You may ask your colleagues whether they use **coding conventions**. You may also read the official C# code conventions from Microsoft: http://msdn.microsoft.com/en-us/library/vstudio/ff926074.aspx.

6. Review all the learned concepts from this chapter and apply them to the code you are given. First **understand how the code works** and then fix the bugs you discover. The best way to start is by **reformatting the code** and **renaming the identifiers**. Then you may **write unit tests** to enable refactoring without a risk to break something. Then step by step you may **extract methods**, **remove the duplicated code**, and **rewrite pieces of the code** which cannot be refactored. Be sure to **test** after each change.

Chapter 22. Lambda Expressions and LINQ

In This Chapter

In this chapter we will become acquainted with some of the advanced capabilities of the C# language. To be more specific, we will pay attention on how to make **queries to collections**, using **lambda expressions** and **LINQ**, and how to add functionality to already created classes, using **extension methods**. We will get to know the **anonymous types**, describe their usage briefly and discuss lambda expressions and show in practice how most of the built-in **lambda functions** work. Afterwards, we will pay more attention to the LINQ syntax – we will learn what it is, how it works and what queries we can build with it. In the end, we will get to know the meaning of the **keywords in LINQ**, and demonstrate their capabilities with lots of examples.

Extension Methods

In practice, programmers often have to **add new functionality** to already existing code. If the code is available, we can simply add the required functionality and recompile. When a given assembly (**`.exe`** or **`.dll`** file) has already been compiled, and the source code is not available, a common way to extend the functionality of the types is trough inheritance. This approach can be quite difficult to apply, due to the fact that we will have to change the instances of the base class with the instances of the derived one to be able to use our new functionality. Unfortunately, that is the least of our problems. If the type we want to inherit is marked with the keyword **`sealed`**, inheritance is not possible.

Extension methods solve that very same problem – they present to us the opportunity to **add new functionality to already existing type** (class or interface), without having to change its original code or use inheritance, i.e. also works fine with types that cannot be inherited. Notice that trough extension methods we can add "implemented methods" even to interfaces.

The extension methods are defined as **`static`** in ordinary static classes. The type of their first argument is the class (or the interface) they extend. In front of it, we should place **the keyword `this`**. That is what makes them different from other static methods, and indicates the compiler that this is an extension method. The parameter with the keyword **`this`** in front of it can be used in the method body to create its functionality. Practically, it is the object that is used by the extension method.

Extension methods can be applied directly to objects of the class/interface they extend. They can also be invoked statically through the static class they are defined in, but it is not a good practice.

	To refer to a specific extension method, we should add "using" and the corresponding namespace, where the static class, describing this method, is defined. Otherwise the compiler has no way of knowing about their existence.

Extension Methods – Examples

Let's take for example the **definition of an extension method** that counts the number of words in a given string. Have in mind, that the type **string** is **sealed**, so it cannot be inherited.

```
public static class StringExtensions
{
  public static int WordCount(this string str)
  {
    return str.Split(new char[] { ' ', '.', '?', '!' },
      StringSplitOptions.RemoveEmptyEntries).Length;
  }
}
```

The method **WordCount(…)** extends the class **String**. This is indicated by the keyword **this** before the type and the name of the first argument of the method (in our case **str**). The method itself is **static** and it is defined in the static class **StringExtensions**. The usage of the extension method is done the same way as all the other methods of the class **String**. Do not forget to add the corresponding namespace, where the static class, describing the extension methods, is defined. Example of **using an extension method**:

```
static void Main()
{
  string helloString = "Hello, Extension Methods!";
  int wordCount = helloString.WordCount();
  Console.WriteLine(wordCount);
}
```

The method is invoked on the object **helloString**, which is of type **string**. It also takes the object as an argument and works with it (in our case refers to its **Split(…)** method and returns the number of elements of the array, produced by the **Split(…)** method).

Extension Methods for Interfaces

Extension methods can not only be used on classes, but on interfaces as well. Our next example takes an instance of a class, that implements the interface list of integers (**IList<int>**), and increases their value by a certain number. The method **IncreaseWith(…)** can access only those elements that are included in the interface **IList** (e.g. the property **Count**).

```
public static class IListExtensions
{
   public static void IncreaseWith(
      this IList<int> list,  int amount)
   {
      for (int i = 0; i < list.Count; i++)
      {
         list[i] += amount;
      }
   }
}
```

The extension methods also give us the opportunity to work on generic types. Let's take for example a method that loops trough a collection, using **foreach**, implementing **IEnumerable** from generic type **T**. Its purpose is to convert to a meaningful string a sequence of elements (e.g. a list of integers):

```
public static class IEnumerableExtensions
{
   public static string ToString<T>(
      this IEnumerable<T> enumeration)
   {
      StringBuilder result = new StringBuilder();
      result.Append("[");

      foreach (var item in enumeration)
      {
         result.Append(item.ToString());
         result.Append(", ");
      }

      if (result.Length > 1)
         result.Remove(result.Length - 2, 2);
      result.Append("]");
      return result.ToString();
   }
}
```

Example of how to use the two extension methods declared above:

```
static void Main()
{
    List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
    Console.WriteLine(numbers.ToString<int>());
    numbers.IncreaseWith(5);
    Console.WriteLine(numbers.ToString<int>());
}
```

The **output** of the execution of the program will be the following:

```
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
```

Anonymous Types

In object-oriented languages (such as C#), it is common to define small classes that will be used only once. Typical example is the class **Point** that has only two fields – the coordinates of a point. Creating a simple class with the idea of using it just once is inconvenient and time consuming for the programmer, especially when the standard operations for each class: **ToString()**, **Equals()** and **GetHashCode()** have to be predefined.

In C# there is a built-in way to create **single-use types**, called **anonymous types**. Objects of such type are created almost the same way as other objects in C#. The thing with them is that we don't need to define data type for the variable in advance. The **keyword var** indicates to the compiler that the type of the variable will be automatically detected by the expression, after the equals sign. We actually don't have a choice here, since we can't tell the specific type of the variable, because it is defined as one of an **anonymous type**. After that, we specify name for the object, followed by the "**=**" operator and the keyword **new**. In curly braces we enumerate the names and the values of the properties of the anonymous type.

Anonymous Types – Example

Here is an **example of creating an anonymous type** that describes a car:

```
var myCar = new { Color = "Red", Brand = "BMW", Speed = 180 };
```

During compilation, the compiler will create a class with a **unique name** (something like **<>f__AnonymousType0**) and will generate properties for it (with getter and setter). In the example above, the compiler will guess by its own, that the properties **Color** and **Brand** are of type **string** and **Speed** will be set as **int**. Right after the initialization, the object of the anonymous type can be used as one of an ordinary type with its three properties:

```
Console.WriteLine("My car is a {0} {1}.",
  myCar.Color, myCar.Brand);
Console.WriteLine("It runs {0} km/h.", myCar.Speed);
```

The output of the code above will be as follows:

```
My car is a Red BMW.
It runs 180 km/h.
```

More about Anonymous Types

As any other type in .NET, the anonymous ones inherit the class **System.Object**. During compilation, the compiler will automatically redefine the methods **ToString()**, **Equals()** and **GetHashCode()** for us.

```
Console.WriteLine("ToString: {0}", myCar.ToString());
Console.WriteLine("Hash code: {0}",
  myCar.GetHashCode().ToString());
Console.WriteLine("Equals? {0}", myCar.Equals(
  new { Color = "Red", Brand = "BMW", Speed = 180 }
));
Console.WriteLine("Type name: {0}", myCar.GetType().ToString());
```

The output of the code above will be the following:

```
ToString: { Color = Red, Brand = BMW, Speed = 180 }
Hash code: 1572002086
Equals? True
Type name:
<>f__AnonymousType0`3[System.String,System.String,System.Int32]
```

As we can see from the result, the method **ToString() is redefined**, so that it can list the properties of the anonymous type in the order of their definition in the initialization of the object (in our case **myCar**). The method **GetHashCode()** is wrote in such a way, that it uses all fields and on their basis it calculates a hash function with a small number of collisions. The redefined by the compiler method **Equals(…)** compares the objects field by field. As we can notice from the example, we have created a new object that has exactly the same properties as **myCar**, and returns a result stating that the newly created object and the old one have equal values.

Arrays of Anonymous Types

The anonymous types, like ordinary ones, can be used as **elements of arrays**. We can initialize them with the keyword **new**, followed by square brackets. The values of the elements of the array are listed the same way, as

the values assigned to the anonymous types. The values in the array should be homogeneous, i.e. it is not possible to have different anonymous types in the same array. An example of defining an array of anonymous types with two properties (**X** and **Y**):

```
var arr = new[] {
   new { X = 3, Y = 5 },
   new { X = 1, Y = 2 },
   new { X = 0, Y = 7 }
};
foreach (var item in arr)
{
   Console.WriteLine(item.ToString());
}
```

The result of the execution of the code above will be the following:

```
{ X = 3, Y = 5 }
{ X = 1, Y = 2 }
{ X = 0, Y = 7 }
```

Lambda Expressions

Lambda expressions are anonymous functions that contain expressions or sequence of operators. All lambda expressions use the lambda operator **=>**, which can be read as "**goes to**". The idea of the lambda expressions in C# is borrowed from the functional programming languages (e.g. **Haskell**, **Lisp**, **Scheme**, **F#** and others). The left side of the lambda operator specifies the **input parameters** and the right side holds an **expression** or a code block that works with the entry parameters and conceivably returns some result.

Usually lambda expressions are used as **predicates** or instead of **delegates** (a type that references a method instance), which can be applied on collections, processing their elements and/or returning a certain result.

Lambda Expressions – Examples

As an example, let's take the **extension method FindAll(…)**, which can be used to filter the necessary elements. It works on a certain collection by applying a given **predicate** on it that checks if an element matches a certain requirement. In order to use it we have to add a reference to the assembly **System.Core.dll** (if it is not already added) and include the namespace **System.Linq**, because the extension methods for the collections are there.

For example, if we want to take only the even numbers from a collection of integers, we can use the method **FindAll(…)** on that collection, passing a lambda method to it that checks if a certain number is even:

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);

foreach (var num in evenNumbers)
{
  Console.Write("{0} ", num);
}

Console.WriteLine();
```

The result is:

```
2 4 6
```

The example above loops through the whole collection of numbers and for each element (named **x**) a check, if the number is multiple of 2, is made (through the Boolean expression **(x % 2) == 0**).

Let's now focus on an example in which trough an **extension method** and a **lambda expression** we will create a collection, containing data from a certain class. In the example, from the class **Dog** (with properties **Name** and **Age**), we want to get a list that contains all dogs' names. We can do that with the **extension method Select(…)** (defined in the namespace **System.Linq**) by assigning to it to turn each dog (**x**) into dog's name (**x.Name**) and writing that result in the variable **names**. With the keyword **var**, we tell the compiler to define the type of the variable according to the result that we assign on the right side of the equals sign.

```
class Dog
{
  public string Name { get; set; }
  public int Age { get; set; }
}

static void Main()
{
  List<Dog> dogs = new List<Dog>() {
    new Dog { Name = "Rex", Age = 4 },
    new Dog { Name = "Sean", Age = 0 },
    new Dog { Name = "Stacy", Age = 3 }
  };
  var names = dogs.Select(x => x.Name);
  foreach (var name in names)
  {
    Console.WriteLine(name);
  }
```

```
}
```

The result is:

```
Rex
Sean
Stacy
```

Using Lambda Expressions with Anonymous Types

We can create **collections of anonymous types** from a collection with some elements by **using lambda expressions**. Let's take the collection **dogs**, containing elements of type **Dog**, and create new collection consisting of elements of an anonymous type, having two properties – age and the initial letter of the dog's name:

```
var newDogsList = dogs.Select(
  x => new { Age = x.Age, FirstLetter = x.Name[0] });
foreach (var item in newDogsList)
{
  Console.WriteLine(item);
}
```

The result is:

```
{ Age = 4, FirstLetter = R }
{ Age = 0, FirstLetter = S }
{ Age = 3, FirstLetter = S }
```

As it is obvious from the example above, the newly created collection **newDogsList** has elements of an anonymous type, taking the properties **Age** and **FirstLetter** as parameters. The first line of the example can be read as follows: "Create a variable of undefined (at this point) type, name it **newDogsList** and create a new element of an anonymous type for each element **x** of the **dogs** collection with two properties: **Age** that is equal to the property **Age** of the element **x**, and the property **FirstLetter** that is equal to the first character of the string **x.Name**".

Sorting with Lambda Expressions

If we want to sort the elements in a certain collection, we can use the extension methods **OrderBy(…)** and **OrderByDescending(…)**, by defining the way of sorting in a lambda function. An example on our collection **dogs**:

```
var sortedDogs = dogs.OrderByDescending(x => x.Age);
```

```
foreach (var dog in sortedDogs)
{
  Console.WriteLine(string.Format(
    "Dog {0} is {1} years old.", dog.Name, dog.Age));
}
```

The result is:

```
Dog Rex is 4 years old.
Dog Stacy is 3 years old.
Dog Sean is 0 years old.
```

Statements in Lambda Expressions

Lambda functions can also have a **body**. So far we have used lambda functions with only one statement. Now we will pay more attention to lambda functions that have a body. Let's return to the example with the even numbers. Suppose we want to print to the console the values of all numbers, to which our lambda function is applied to and to return the result if they are even or not. We can do it the following way:

```
List<int> list = new List<int>() { 20, 1, 4, 8, 9, 44 };
// Process each argument with code statements
var evenNumbers = list.FindAll((i) =>
{
  Console.WriteLine("Value of i is: {0}", i);
  return (i % 2) == 0;
});
```

The result from the above code is:

```
Value of i is: 20
Value of i is: 1
Value of i is: 4
Value of i is: 8
Value of i is: 9
Value of i is: 44
```

Lambda Expressions as Delegates

Lambda functions can be written in delegates. **Delegates** are such a type of variables that contains functions (methods). Some standard delegate types in .NET are: **`Action`**, **`Action<in T>`**, **`Action<in T1, in T2>`**, and so on and **`Func<out TResult>`**, **`Func<in T, out TResult>`**, **`Func<in T1, in T2, out TResult>`** and so on. The types **`Func`** and **`Action`** are generic and

contain the types of the return value, and the types of the parameters of the functions. The variables of such types are references to functions. Below is an example for using and assigning values to these types:

```
Func<bool> boolFunc = () => true;
Func<int, bool> intFunc = (x) => x < 10;
if (boolFunc() && intFunc(5))
{
  Console.WriteLine("5 < 10");
}
```

The result is:

```
5 < 10
```

In the example above we define **two delegates**. The first one – **boolFunc** is a function that has no input parameters and returns a Boolean result. We have given an anonymous lambda function that does nothing and always returns **true** as a value to that function. The second delegate **intFunc** takes as an argument an **int** variable and returns a Boolean value – true when **x** is less than ten, and false otherwise. At the end, in the **if** statement, we call these two delegates as we give to the second one value of 5 as an argument, and the result from their invocation is **true**, as we can see.

LINQ Queries

LINQ (Language-Integrated Query) is a set of extensions of the .NET Framework, that includes language integrated queries and operations on the elements of a certain **data source** (most often arrays or collections). LINQ is a **very powerful tool**, similar to most SQL languages by logic and syntax. It actually works with collections in the same way as SQL languages work with table rows in databases. It is part of the syntax of C# and Visual Basic .NET and consists of few special keywords like **from**, **in** and **select**. In order to use LINQ queries in C#, we have to include a **reference to System.Core.dll** and to **include the namespace System.Linq** in the beginning of the C# program.

Data Sources with LINQ

To define the data source (collection, array and so on), we have to use the keywords **from** and **in** and a variable for the iteration of the collection (the iteration is similar to the one with the **foreach** operator). For example, a query that starts like this:

```
from culture
in CultureInfo.GetCultures(CultureTypes.AllCultures)
```

can be read as follows: "for each element of the collection **CultureInfo.GetCultures(CultureTypes.AllCultures)** assign the variable **culture** and use it to refer to these items further in the query".

Data Filtering with LINQ

The keyword **where** can be used to set conditions, that should be kept by each item of the collection, in order to continue with the execution of the query. The expression after **where** is always of a Boolean type. We can say that **where works as a filter for the elements**. For example, if we want to see only those cultures, whose name begins with the lowercase Latin letter **b**, we can continue the query from our last example like this:

```
where culture.Name.StartsWith("b")
```

As we can notice, after **where … in**, we use only the name we gave for the iteration of the variables in the collection. The keyword **where** is compiled up to the invoking of the extension method **Where()**.

```
where culture.Name.StartsWith("b")
```

Results of LINQ Queries

To **choose the output data** for the query, we can **use the keyword select**. The result is an object of an existing class or an anonymous type. The result can also be a property of the objects, the query runs through or the objects themselves. The **select** statement and everything following it is placed **always at the end of the query**. The four keywords: **from**, **in**, **where** and **select**, are completely enough to create a simple LINQ query. Here is an example:

```
List<int> numbers = new List<int>() {
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
var evenNumbers =
  from num in numbers
  where num % 2 == 0
  select num;
foreach (var item in evenNumbers)
{
  Console.Write(item + " ");
}
```

The result is:

```
2 4 6 8 10
```

The example above runs a **query over a collection of integers called numbers** and filters only the even ones in a new collection. The query can be read as follows: "for each number **num** from **numbers** check if it is multiple of 2, and if so, add it to the new collection".

Sorting Data with LINQ

Sorting with LINQ queries is done through the **keyword orderby**. The conditions, used for sorting the elements, are placed after it. For each condition the order of arrangement can be indicated: ascending (using the keyword **ascending**) and descending (with the keyword **descending**), as by default the elements are ordered in ascending order. If we want to sort an array of strings by their length in descending order, for example, we can write the following query:

```
string[] words = { "cherry", "apple", "blueberry" };
var wordsSortedByLength =
   from word in words
   orderby word.Length descending
   select word;
foreach (var word in wordsSortedByLength)
{
   Console.WriteLine(word);
}
```

The result is:

```
blueberry
cherry
apple
```

If no instruction for the order is given (i.e. the keyword **orderby** is missing from the query) the items are printed in the way they would be processed, if the **foreach** operator was used.

Grouping Results with LINQ

To group the results by some criteria the keyword **group** should be used. The pattern is as follows:

```
group [variable name] by [grouping condition] into [group name]
```

The **result of grouping is a new collection of a special type** that can be used further in the query. After the grouping, however, the query stops working with its initial variable. This means that in the **select** statement, we can use only the group. An example of grouping:

```
int[] numbers =
  { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0, 10, 11, 12, 13 };
int divisor = 5;

var numberGroups =
  from number in numbers
  group number by number % divisor into group
  select new { Remainder = group.Key, Numbers = group };

foreach (var group in numberGroups)
{
  Console.WriteLine(
  "Numbers with a remainder of {0} when divided by {1}:",
    group.Remainder, divisor);
  foreach (var number in group.Numbers)
  {
    Console.WriteLine(number);
  }
}
```

The result is:

```
Numbers with a remainder of 0 when divided by 5:
5
0
10
Numbers with a remainder of 4 when divided by 5:
4
9
Numbers with a remainder of 1 when divided by 5:
1
6
11
Numbers with a remainder of 3 when divided by 5:
3
8
13
Numbers with a remainder of 2 when divided by 5:
7
2
12
```

As we can see from the example above, the numbers printed to the console are grouped by their remainders of the division by 5. In the query, for each number **number % divisor** is calculated, and for each different result a new

group is formed. Further in the query, the **select** operator works on the list of created groups, and for each group creates an anonymous type with two properties: **Remainder** and **Numbers**. To the property **Remainder** the **key of the group** is assigned (in our case the remainder of the division by the **divisor** of the number). And to the property **Numbers** the collection **group** is assigned, that contains all the elements in the group. Notice that **select** is executed only over the list of groups. The variable **number** cannot be used there. Further in the example of two nested **foreach** statements, the remainders (the groups) and the numbers that have the remainder (located in the group) are printed.

Joining Data with LINQ

The **join** statement is a bit more complicated than the other LINQ statements. It joins collections by certain matching criteria and extracts the needed data. Its syntax is as follows:

```
from [variable name from collection 1] in [collection 1] join
[variable name from collection 2] in [collection 2] on [part of
the compare condition from collection 1] equals [part of the
compare condition from collection 2]
```

Further in the query (e.g. in the **select** part), both, the name of the variable from collection 1, and the name of the variable from collection 2, can be used. Example:

```
public class Product
{
  public string Name { get; set; }
  public int CategoryID { get; set; }
}

public class Category
{
  public int ID { get; set; }
  public string Name { get; set; }
}
```

The code that illustrates how to use LINQ joins:

```
List<Category> categories = new List<Category>()
{
  new Category() { ID = 1, Name = "Fruit" },
  new Category() { ID = 2, Name = "Food" },
  new Category() { ID = 3, Name = "Shoe" },
  new Category() { ID = 4, Name = "Juice" },
```

```
};
List<Product> products = new List<Product>()
{
   new Product() { Name = "Strawberry", CategoryID = 1 },
   new Product() { Name = "Banana", CategoryID = 1 },
   new Product() { Name = "Chicken meat", CategoryID = 2 },
   new Product() { Name = "Apple Juice", CategoryID = 4 },
   new Product() { Name = "Fish", CategoryID = 2 },
   new Product() { Name = "Orange Juice", CategoryID = 4 },
   new Product() { Name = "Sandal", CategoryID = 3 },
};
var productsWithCategories =
   from product in products
   join category in categories
      on product.CategoryID equals category.ID
   select new {  Name = product.Name,
                 Category = category.Name };
foreach (var item in productsWithCategories)
{
   Console.WriteLine(item);
}
```

The result is:

```
{ Name = Strawberry, Category = Fruit }
{ Name = Banana, Category = Fruit }
{ Name = Chicken meat, Category = Food }
{ Name = Apple Juice, Category = Juice }
{ Name = Fish, Category = Food }
{ Name = Orange Juice, Category = Juice }
{ Name = Sandal, Category = Shoe }
```

In the example above, we create two classes and an imaginary relationship between them. To each product some category **CategoryID** (represented by a number) corresponds, that matches the number **ID** from the class **Category** in the collection **categories**. If we want to use this relation and to create a new anonymous type, where to store the products and their names and category, we can write the above LINQ query. It joins the collection of elements of type **Category** with the one of type **Product** by the mentioned criteria (match between **ID** from **Category** and **CategoryID** from **Products**). In the **select** part of the query, we use both names **category** and **product** to construct an anonymous type with the name of the product and the name of the category.

Nested LINQ Queries

LINQ also supports **nested queries**. For example our last query can be written by nesting two queries in the following way (the result is exactly the same as the one with **join**):

```
var productsWithCategories =
  from product in products
  select new {
    Name = product.Name,
    Category =
      (from category in categories
      where category.ID == product.CategoryID
      select category.Name).First()
  };
```

Since each query in LINQ returns a collection of items (irrespective of whether the result from it is of 0, 1 or more elements), we need to use the extension method **First()** over the result of the nested query. The method **First()** returns the first element (in our case the only one) of the collection it is applied on. In this way we get the name of the category only by its ID number.

LINQ Performance

As a rule using **LINQ and extension methods is slower than using direct operations** over a collection of elements, so beware of using LINQ when processing large collections or the performance is critical.

Let's compare the speed of adding 50,000,000 elements to a list through extension methods and directly with a **for**-loop:

```
List<int> l1 = new List<int>();
DateTime startTime = DateTime.Now;
l1.AddRange(Enumerable.Range(1, 50000000));
Console.WriteLine("Ext.method:\t{0}", DateTime.Now - startTime);

startTime = DateTime.Now;
List<int> l2 = new List<int>();
for (int i = 0; i < 50000000; i++) l2.Add(i);
Console.WriteLine("For-loop:\t{0}", DateTime.Now - startTime);
```

The result might be as follows (depends on the computer's CPU speed):

```
Ext.method:     00:00:01.6430939
For-loop:       00:00:00.9120522
```

LINQ technology and extension methods work through the concept of **expression trees**. Each LINQ query is translated by the compiler to an expression tree and is executed when its results are actually accessed (not earlier). For example let's consider the following code:

```
List<int> list = new List<int>();
list.AddRange(Enumerable.Range(1, 100000));

DateTime start = DateTime.Now;
for (int i = 0; i < 10000; i++)
{
  var elements = list.Where(e => e > 20000);
}
Console.WriteLine("No execution:\t{0}", DateTime.Now - start);

start = DateTime.Now;
for (int i = 0; i < 10000; i++)
{
  var element = list.Where(e => e > 20000).First();
}
Console.WriteLine("Execution:\t{0}", DateTime.Now - start);
```

The result might be as follows (depends on the computer's CPU speed):

```
No execution:   00:00:00.0070004
Execution:      00:00:02.7231558
```

This shows that if we call a **.Where(…)** filter (or **where** clause in LINQ) it is not actually executed until its result is actually needed. The elements **get filtered on demand**, at the time they are really required. In our case this is when we invoke **First()** method. Moreover, if we get the first element of a sequence, the rest elements are not processes until needed. Thus if we use change the filtering lambda function from "**e => e > 20000**" to "**e => e > 500000**", the filtering becomes times slower because more elements are processed until the first matching the filtering condition is found:

```
No execution:   00:00:00.0060004
Execution:      00:00:06.3663641
```

Standard .NET Framework collection classes like **List<T>**, **HashSet<T>** and **Dictionary<K,V>** are optimized to work fast with LINQ. Most operations with LINQ work almost as fast as if we run them directly. Let's check this example:

```
HashSet<Guid> set = new HashSet<Guid>();
for (int i = 0; i < 50000; i++)
{
```

```
    set.Add(Guid.NewGuid()); // Add random GUID
}

Guid keyForSearching = new Guid();
DateTime start = DateTime.Now;
for (int i = 0; i < 50000; i++)
{
    // Use HashSet.Contains(…)
    bool found = set.Contains(keyForSearching);
}
Console.WriteLine("HashSet: {0}", DateTime.Now - start);

start = DateTime.Now;
for (int i = 0; i < 50000; i++)
{
    // Use IEnumerable<Guid>.Contains(…) extension method
    bool found = set.Contains<Guid>(keyForSearching);
}
Console.WriteLine("Contains: {0}", DateTime.Now - start);

start = DateTime.Now;
for (int i = 0; i < 50000; i++)
{
    // Use IEnumerable<Guid>.Where(…) extension method
    bool found = set.Where(g => g==keyForSearching).Count() > 0;
}
Console.WriteLine("Where: {0}", DateTime.Now - start);
```

The result is as follows (though it depends on the computer's CPU speed):

```
HashSet: 00:00:00.0030002
Contains: 00:00:00.0040003
Where: 00:02:49.9717218
```

Seems like .NET Framework takes into account the capability to search in constant time O(1) in a **HashSet<T>**, so searching though the native method **Contains(…)** and though the extension methods **IEnumerable.Contains(…)** both **run in time O(1)**. By contrast, the **IEnumerable.Where(…)** method is **dramatically slower** and runs in linear time O(n). This is expected, because the **Where(…)** method checks certain condition for each element in a collection and it is expected to process all elements one by one. By contrast the **Contains(…)** method just searches for single element which is fast operation.

In case you do not remember about the asymptotic notation O(1) and O(n), please check the chapter "Data Structures and Algorithm Complexity".

In the above example we use the system structure **Guid**. This is a global unique identifier often used in computer technologies to identify an object. It may look like the following: **8668f585-faf8-4685-8025-6a8d1d2aba0a**. If you want to generate a global unique (world-wide) identifier, you might benefit from the method **Guid.NewGuid()**, like we do in the code above.

Exercises

1. Implement an extension method **Substring(int index, int length)** for the class **StringBuilder** that returns a new **StringBuilder** and has the same functionality as the method **Substring(…)** of the class **String**.

2. Implement the following extension methods for the classes, implementing the interface **IEnumerable<T>**: **Sum()**, **Min()**, **Max()**, **Average()**.

3. Write a class **Student** with the following properties: first name, last name and age. Write a method that for a given array of students finds those, whose first name is before their last one in alphabetical order. Use LINQ.

4. Create a **LINQ query** that finds the first and the last name of all students, aged between 18 and 24 years including. Use the class **Student** from the previous exercise.

5. By using the extension methods **OrderBy(…)** and **ThenBy(…)** with **lambda expression**, **sort a list of students** by their first and last name in descending order. Rewrite the same functionality using a **LINQ query**.

6. Write a program that prints to the console all numbers from a given array (or list), that are **multiples of 7 and 3 at the same time**. Use the built-in **extension methods** with **lambda expressions** and then rewrite the same using a **LINQ query**.

7. Write an extension method for the class **String** that **capitalizes all letters**, which are the beginning of a word in a sentence in English. For example: "**this iS a Sample sentence.**" should be converted to "**This Is A Sample Sentence.**".

8. Create a hash-table to hold a **phone book**: a set of person names and their phone numbers (e.g. Kate Wilson → +3592981981, +3598862536; Alex & Co. → 1-800-ALEX; Steve Milton → +496023456). Fill enough random data (e.g. 50,000 key-value pairs). Measure **how much time it takes** to perform searching by key in the hash-table using its native search capabilities, using the extension methods **IEnumerable.Contains(…)** and **IEnumerable.Where(…)**. Can you explain the difference?

Solutions and Guidelines

1. Follow the syntax explained in the **section "Extension Methods"**. You may create a new **StringBuilder** and to write in it all the characters with indices, starting from **index** and with length **length**, from the object that the extension method will work on.

2. For generic implementation of the **Min()** and **Max()** methods for any generic type **T** you can add a restriction to the passed type **T** to be **comparable**, i.e. you should have something like this:

```
public static T Min<T>(this IEnumerable<T> elements)
  where T : IComparable<T>
{ … }
```

   Since not all data types have predefined operators **+** and **/**, it will not be possible to apply the functions **Sum()** and **Average()** to all types directly. There are no interfaces **ISummable<T>** and **IDividable<T>** in .NET. One way to work around this problem is to **convert all input objects to decimal** and then to calculate sum / average and return **decimal** as result. For the conversion you can use the static method **Convert.ToDecimal(…)**.

   Another interesting approach is to use the **dynamic data type** in C# to hold the arguments and results and to execute the operations over them **at runtime** (due to the dynamic evaluation capabilities in C#):

```
public static dynamic Min<T>(this IEnumerable<T> elements)
{ … }
```

   This is **easier to implement** and works better but could have performance issues and some special cases to be handled.

3. Review the keywords **from**, **where** and **select** from the "LINQ Queries" section in this chapter.

4. Write a **LINQ query** to select the described students in an **anonymous type** that contains only two properties – **FirstName** and **LastName**.

5. For the **LINQ query** use **from**, **orderby**, **descending** and **select**. For the implementation with the **lambda expressions**, you can use the methods **OrderByDescending(…)** and **ThenByDescending(…)**.

6. It is enough to check if the numbers are multiples of 21, instead of writing two **where** conditions.

7. Use the method **ToTitleCase(…)** of the property **TextInfo** in the culture **en-US** in the following way:

```
new CultureInfo("en-US", false).TextInfo.ToTitleCase(text);
```

8. See the examples at the end of the section "LINQ Performance". You can use **Dictionary<string, List<string>>** to hold the phone book. You may **explain the difference in the execution speed** by trying to explain how searching works internally and by the assumption that searching in a hash-table takes time O(1) and searching in a collection element by element runs in linear time O(n).

Chapter 23. Methodology of Problem Solving

In This Chapter

In this chapter we will discuss one **recommended practice for efficiently solving computer programming problems** and make a demonstration with appropriate examples. We will discuss the basic engineering principles of **problem solving**, why we should follow them when solving computer programming problems (the same principles can also be applied to find the solutions of many mathematical and scientific problems as well) and we will make an example of their use. We will **describe the steps**, in which we should go in order to solve some sample problems and show the mistakes that can occur when we do not follow these same steps. We will pay attention to some important steps from the methodology of problem solving, that we usually skip, e.g. the **testing**. We hope to be able to prove you, with proper examples, that the solving of computer programming problems has a "recipe" and it is very useful.

Basic Principles of Solving Computer Programming Problems

You probably think this chapter is about an idle talk like "first think, then act" or "be careful when you write and try to not miss something". In fact this chapter will not be so tedious and boring and will give you some **practical guidelines for solving algorithmic problems** as well as other problems.

Without making any claim of completeness, we will give you some important suggestions, based on **Svetlin Nakov's personal experience** acquired during his work of 10+ years as a competitor in International and Bulgarian programming competitions. Svetlin has gained tens of **International awards** from programming contests including medals from International Olympiad in Informatics (IOI) and has been training students from **Sofia University** St. Kliment Ohridski (SU), **New Bulgarian University** (NBU), **Technical University of Sofia** (TU-Sofia), **National Academy for Software Development** (NASD), and **Telerik Software Academy**, and his experience during the last 10 years confirms that this methodology works well in practice.

Let's start with the first key suggestion.

Use Pen and Paper

The **use of a pen and sheet of paper** and the making of drafts and sketches when solving problems is something normal and natural, which every experienced mathematician, physicist and software engineer does when tasked with a non-trivial problem.

Unfortunately, our experience with students showed us most of the novice programmers **do not even bring with them a pen and paper**. They have the false perception that in order to solve programming problem they only need a keyboard. Most of them need some time and exams' failures to finally realize that the **making of some kind of drafts on paper is crucial** for understanding the problem and constructing a correct solution.

	Everyone who does not use a pen and paper will be in a serious trouble when solving computer programming problems. It is important always to make drafts of your ideas on paper or blackboard before even start typing on the keyboard.

Maybe, it is a little old-fashioned, but **the "era of the paper" is not over yet**! The easiest way for you to visualize your idea is to put it on paper. It is very difficult for most people to try and think about a problem without some kind of **visualization**. The **visual system in the human brain**, which absorbs information, is strongly connected to these parts of the brain, which are responsible for the creative potential and logical thinking.

People who have well-developed their **visual system** in the brain are able to easily "see" the solution of a problem in their mind. Then they only have to polish their idea and implement it. These people actively use their visual memory and their ability to **create visual imagery**, which is the reason why they can quickly create ideas and reflect on algorithms for solving problems. These people can quickly **recognize and discard the wrong ideas** and **visualize the correct algorithm** for the programming problem in a matter of seconds. Regardless of whether you are a "visual" type of person or not, writing down and sketching your idea is very useful and will most certainly help your thoughts on the matter. **Most people** have the ability to easily present information to the brain visually.

Think for example, how hard it is for you to **multiply five digit numbers in your head** and how less effort does it cost when you **use a pen and paper** (we eliminate the possibility of using electronic calculating devices, of course). It is basically the same with problem-solving, when you need a **clear view on the problem** you should use pen and paper. When you need to check for flaws in your algorithm, you should make some calculations using a pen and paper. When you need to think about a case in which your algorithm might not work, **you should use pen and paper**. That's why you should always use pen and paper!

Generate Ideas and Give Them a Try!

As we have mentioned previously, the first thing to do is to sketch some sample examples for the problem on a piece of paper. When we have a **real example of the problem** in front of us, we can reflect on it and the ideas come.

When the idea is a fact, we need more examples in order to **check if it is a good one**. Then we need some more examples, drafted on paper to verify it again. We should be completely sure our solution is correct. Then we should go through our solution one more time, step by step, the same way like one actual computer program would do, and see if everything runs correctly.

The next thing to do is to **try "breaking" our solution** and thinking of a case, in which our idea would not work properly (a **counter-example**). If we fail at that, then our idea is probably right. If our solution definitely has a flaw, we should think of a way to fix it. If our idea does not pass every test, we should invent a new one. Not always **the first idea that comes** to your mind **is the right one** and is a true solution of the problem.

Problem-solving is an **iterative process**, which represents the **invention of ideas and then testing them** over different examples until you reach one, which seems to work correctly with every example that you could think of.

Sometimes it **can take hours for you to try and find the right solution** of a given problem. This is completely normal. Nobody has an ability to instantly find the correct solution of a problem, but surely the more experience you have the faster the good ideas will come. If a particular problem has something in common with one that you have solved in the past, then the proper idea will come to your mind more quickly, because one of the basic characteristics of the human brain is to **work with analogies**. The experience you get from solving given type of problems will help you with the invention of ideas for a solution of other analogical problems.

In order to generate ideas and test them it is mandatory **to have a piece of paper, pen and different examples**, which you need to visualize with the help of drafts, sketches or other means. That can help you a lot to quickly try different ideas and reflect on the solutions, which can occur to you. The basic things you need to do when you solve problems is to logically think of some problems that are analogical to the current one, summarize or try to use general ideas and then construct your solution using pen and paper. When you have a **sketch in front of you** it is easier to imagine what could possibly go wrong. This might give you an idea for the next step or make you give up your current idea entirely. In this way we can get a complete algorithm, the correctness that can be tested by a specific example.

The problem solving starts with the invention of ideas and testing them. This is best done with a pen and paper in hand and sample sketches and drafts to help you think. Always test your ideas and solutions with proper examples!

The recommendations given above are also very useful in one more case – when you are **at a job interview**. Every experienced interviewer could agree, that when he gives an algorithmic problem to the interviewee he expects from him **to take a pen and piece of paper**, to reflect on the problem out loud and to give different suggestions for the solution. This is a sign this person can think and has a **proper approach to the problem solving**. Thinking out loud and rejecting different ideas shows that the interviewee has the right thinking. Even if he fails to solve the problem, this behavior will make a good impression to the interviewer!

Decompose the Task into Smaller Subtasks

Complex tasks can always be **divided into smaller more manageable subtasks**. We will show this with some examples below. There is not a single complex problem in this world that has been solved with one try. The correct formula for solving such a task is to **split it into smaller simpler tasks**, which have to be independent and different from one another. If these smaller subtasks prove to be complicated, we should split them again. This technique is called "**divide and conquer**" and it is in use since the time of the Roman Empire.

The division of the problem into smaller units is easier said than done. The essence of solving algorithmic problems is in the good technique of division of the given task into simpler subproblems and, of course, the invention of good ideas that can be achieved with gaining more experience.

	Complex tasks can always be divided into smaller more manageable subtasks. When you have to solve big complicated tasks, you should always try to divide it into simpler problems, which are easier to solve.

"Cards Shuffle" Problem – Example

Let’s give the following example: we have one **ordered deck of cards** and we have to **shuffle it in random order**. Let’s assume that the deck is represented as an array or list of N objects (every card is an object). These types of tasks require multiple repeating steps (series of removal, placing, replacing and realignment of elements). Each of these steps itself is simpler, easier and **more manageable subtask**, than the "Cards Shuffle" task as a whole. If we succeed in decomposing the complex task into smaller subtasks, we will basically find the right way to solve the problem. Exactly this is **the essence of the algorithmic thinking**: the ability to decompose complex problems into smaller ones and then find the correct solutions for them. Of course, this principle can be applied not only to programming problems, but also to ones from other scientific disciplines like math and physics. In fact this algorithmic thinking is the reason why the mathematicians and the physicists show a rapid progress when they begin to learn computer programming.

Now let's go back to the given task and think about how to find the simple **subtasks**, which are needed in order to meet the requirements to randomly shuffle the cards.

If we take one deck of cards in our hands or try to sketch something on paper (e.g. series of rectangular cells, each of them representing one card), some **ideas instantly come up**, for example we need to change or realign elements from the deck.

Thinking like this, we can easily reach the conclusion we need to make more than one swap of one or more cards. If we make only one swap, the deck of cards would not be completely random. Therefore we need many simpler operations for a **single swap (exchange)**.

We reached the point where we do the first decomposition into smaller subtasks: we need **series of swaps**, which can be considered as smaller tasks, a part of the bigger problem.

First Subtask: a Single Swap

How do we make a single swap of cards in the deck? We can answer this question in many ways and take the first idea that come to our mind. If it is any good, we will use it. Otherwise we will think of something else.

Our **first idea** can be: if we have a deck of cards, we can split it at random card and then separate and swap the two parts. Now do we have an idea for a single swap? Yes, we have. The next thing to do is to check if our solution is working properly (we will demonstrate this after a while).

Now let's go **back to the base task**: after applying our idea, we need the deck of cards to be randomly shuffled. Now we split and swap it many times and check the result. It seems that our algorithm works fine and the subtask "single swap" will do the work.

Second Subtask: Choosing a Random Number

How to generate a random number and use it to split the deck? If we have N cards, we need a random number between 1 and N-1, don't we?

In order to solve this problem we might need an additional help. If we know that in .NET Framework this task is already solved, we can simply use the integrated **random number generator**.

Otherwise we have to think of a solution e.g. we can read one line from the keyboard and then measure the time span between the start of the program and the pressing of the button [Enter]. Since the time of every input is different (especially, if we report with accuracy to nanoseconds), we have a **way to calculate a random number**. The only problem now is to find a way to place this number in the interval [1…N-1] and probably most of us will remember that we can use the remainder of its division by (N-1).

We can see that **even the simplest subtasks can be divided into smaller tasks**, which sometimes can be already solved for us. When we find a

suitable solution for the current subtask, we need to go back to the base problem and test everything and see if it is working correctly put together. Let’s do that now, shall we?

Third Subtask: Combining Swaps

Let’s go back to the main task. We have reached the conclusion we have to make as many "single swap" operations as needed to ensure the deck of cards will be correctly shuffled. This idea seems right and we should try it.

Now this raises the question **how many operations "single swap" are enough**? Are 100 enough? Aren’t they too many? And what about 5 times? In order to give a good answer to this question, we need to think for a while. How many cards do we have? If we have several cards in the deck, we need fewer swaps. And if we have many cards, we need much more swaps, right? Therefore the number of swaps depends on the number of cards in the deck.

To see how many swaps are enough, we can take one standard deck of cards. How many cards are there in one standard deck? Most of us know there are **52 cards** in it. Well then try to figure out how many "single swap" operations are needed to randomly shuffle one deck of 52 cards. Are 52 enough? It seems enough because if we swap 52 times at random position it is likely that we will split the deck at every card (this conclusion is clear even if we do not know anything about Probability and Statistics). **52 "single swap" operations** seem too much, isn’t it? Let’s think of even smaller number. What about the half of the number 52? It seems fine as well, but it would be more difficult to explain why.

Some of you probably think that the best way to find the correct number is to use complex formulas from the probability theory, but does it make any sense? The number **52 is small enough** and there is no need to look for other number. One loop of 52 iterations is fast enough. The cards in the deck would not be billions, would they? Therefore we do not have to think in that direction. We assume that the correct number of "single swap" equals **the number of the cards in the deck** – neither too big nor too small. And this is the end of the current subtask.

Another Example: Sorting Numbers

Let’s think of another example. We are given an **array of numbers** and our task is to **sort it in ascending order**. There is an abundance of algorithms for this problem and some of them conceptually different from one another. Even you could think of some ideas to solve this problem, some of them would be right and others – not quite.

So we have to solve this task and we are not allowed to use built-in .NET Framework sorting methods. The first obvious thing to do is to take a pen and piece of paper and to think of one example and then to reflect on the task. Thus we can invent multiple and very different ideas like:

- **First idea**: we can find the smallest number, print it and then remove it from the array of numbers. The next thing to do is to repeat the same action until the array is empty. Thinking like this, we can decompose this task into simpler tasks: finding the smallest number in array; deleting a number from array; printing a number.
- **Next idea**: we can find the smallest number and put it at the first position of the array (swap operation). Then we can do the same action for the rest of the array. Since we have already placed number on the first position, we go to the next one. If we repeat this k times, we will have the first k smallest numbers from the array at the first k positions. This approach takes us naturally to a task, which can be very easily divided into smaller subtasks: finding the number with the smallest value in a part of the array and exchanging the positions of two numbers from the array. The second subtask can be divided one more time: removing an element from a given position and placing an element at a given position.
- **Another idea**, which uses a method, conceptually different from the previous two solutions: we split the array into two subarrays with approximately the same number of elements. Then we sort them individually and finally we merge them into one. We can do this action recursively with every subarray until every one of them holds exactly one element. Array with one element is a sorted one. Here, like in the previous two ideas, we can divide the complex problem into smaller more manageable problems: splitting one array into two parts with approximately equal number of elements; merging two arrays into one big array.

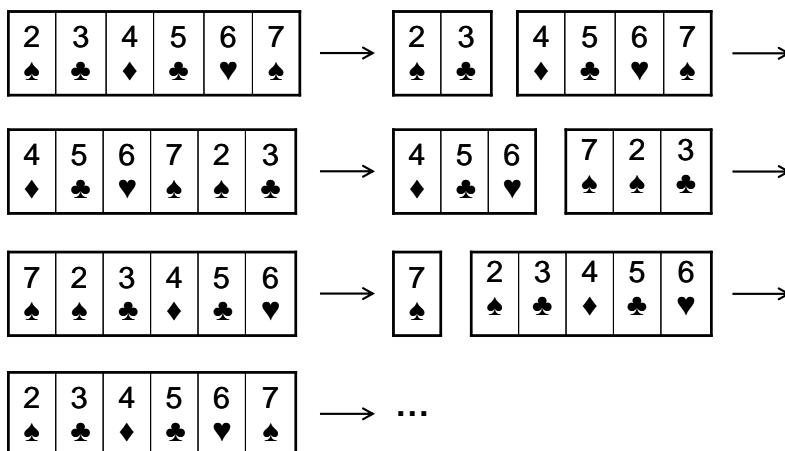
There is no need to continue, right? It is obvious that every one of you can think of several different solutions or you can read about the subject in a book about algorithms. We demonstrated that **every complicated problem can be divided into smaller simpler problems**. These is a correct approach to solving computer programming problems – to think of the big task like it is a collection of smaller easier subtasks. This technique may be hard to learn, but in time you will get used to it.

Verify Your Ideas!

It seems that we have figured out everything. We have an idea. It seems to work properly. The only thing for us to do is to check if our idea is correct or it is only correct in our minds. After that we can start with the implementation.

How to verify an idea? Usually this happens with the help of some examples. We should choose examples that fully cover all different cases, which our algorithm should be able to pass. The sample examples should not be too easy for your algorithm, but also they should not be so hard to be sketched. We call these certain types of examples "**good representatives of the common case**".

E.g. if our task is to sort an array in ascending order, then a suitable example would be an **array with 5-6 elements**. Two of the numbers in the array should be equal and the other – different. The numbers should be randomly placed in the array. This is a good example, because it covers most of the common cases, in which our algorithm should work.

There are **many inappropriate examples** for the sorting numbers problem that could not help you test your idea properly. For example if you use an array of only two elements. Your solution could work correctly with it, but your core idea could be completely wrong. Another inappropriate example is an array of equal numbers. Every sorting algorithm would work correctly with it. And another bad example – we can use an array that is already sorted. Algorithm could also work correctly and yet the idea could be wrong.

	When verifying your ideas, choose your examples carefully. They should be simple and easy enough for you to be able to sketch them down by hand in a minute and at the same time they should represent most general case in which your idea should work. Your examples should be good representatives of the common case and cover as much cases as possible without being too big and complicated.

"Cards Shuffle" Problem: Verifying the Idea

Let's think of one sample example for our "Cards Shuffle" task. Let's say we have 6 cards. In order our example to be good, our deck of cards **should not be too small** (e.g. 2-3 cards), because in this way our example might become very easy. Also, if we want to easily check our idea with the deck, it **should not be too big**. Initially it is a good idea to get **six cards** and order them in the deck. In this way it would be easier for us to see if the cards are well shuffled or partially shuffled or not shuffled at all. So one of the smartest things to do is to choose 6 cards regardless of their suit and order them by value.

Now we already have one example, which is a **good representative of the common case** of our problem. Let's now sketch it down on a piece of paper and check our algorithm on it. We should split the deck into two parts, at a random position 6 times and then swap them. Our cards are ordered by value. At the end we expect them to be randomly shuffled.

Let's see what is going to happen:

There is no need to do 6 swaps. After only 3 swaps we came back to the starting position. This is probably not an accident. What happened? We have **just found an error in our algorithm**. When we reflect on the problem we can see that with every swap at a random position we rotate the deck to left and after N times it goes to the starting position. So it was a good thing that we tested our idea before even started writing some code, wasn't it?

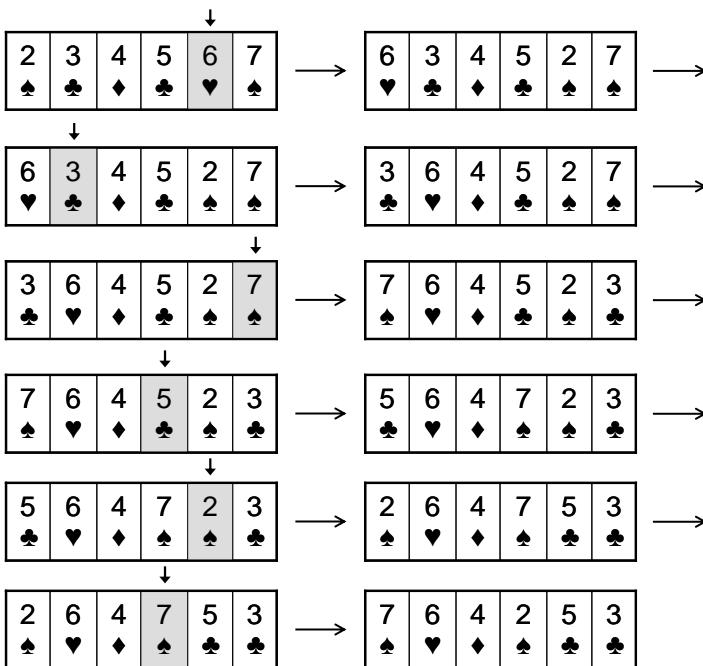
Sorting Numbers: Verifying the Idea

It is time to **check our first idea** considering the sorting numbers problem. We can easily see if it is right or wrong. We start with an array of N elements and we find the smallest number, print it and then delete it from the array N times. Even if we do not sketch the idea, it seems faultless. Still let's think of one example and see what is going to happen. We take 5 numbers, two of them are equal: 3, 2, 6, 1, 2. We have 5 steps to do:

1) 3, 2, 6, 1, 2 → 1
2) 3, 2, 6, 2 → 2
3) 3, 6, 2 → 2
4) 3, 6 → 3
5) 6 → 6

Seems like **our algorithm works properly**. Our result is correct and we do not have a reason to think that our idea will not work with any other example.

If a Problem Occurs, Invent a New Idea!

When you find your idea is incorrect, the obvious thing to do is **to invent a new, better idea**. We can do this in two ways: we can either try to fix our old idea or create a completely new one. Let's see how this works with our cards shuffle problem, shall we?

The creating of a solution for a computer programming problem is an iterative process, which consists of inventing ideas, verifying them and sometimes, when problem occurs, inventing new ones. Sometimes the first idea that comes to our mind is the right one, but most of the times we need to go through many different ideas until we reach the best one.

Let's go back to our **card shuffle problem**. Firstly let's see why our premier idea is wrong and is it possible to fix it? The problem here is easily recognized: the continuous splitting and card swapping does not shuffle them randomly; it simply rotates them to left.

How to fix this algorithm? We need to think of a new and better way to make a "single swap" operation, don't we? Our new idea for one single swap is: randomly choose two cards from the deck and swap their places. If we do this N number of times, we would probably get randomly shuffled deck. This idea looks better than the previous one and maybe it would work correctly this time. We already know that **before we even start thinking of implementing our new algorithm it is better to check it** and see if it is working properly. We can verify our idea by using pen and paper and the example with the 6 cards that we used above.

In this moment we think of an even better idea, instead of choosing 2 random cards from the deck, why not just **pick one random card and swap it with the first card from the deck**? Isn't this idea simpler and easier to implement? The result should be random too. Let's start by choosing a random card at position k_1 and swap it with the first card. Now we have a random card at the first position and the first card is at the k_1 position. On the next step of the algorithm we pick another card at random position k_2 and then swap it with the card from the first position (previously the card from the position k_1). It is apparent that with only 2 steps we have changed the place of the first, the k_1-st and the k_2-nd cards from the deck with random cards. It seems that at every step one card changes its position with a random one. After N number of steps we can expect that **every card from the deck has changed its position** averagely one time. Hence our solution is working and the cards should be well shuffled.

Now **we should test our new idea**. Does it work properly? Let's make sure that what has happened last time will not happen again, shall we? Let's thoroughly check this idea as well. Again, we can take **the 6 cards example, which represents most of the general cases** of the card shuffle problem (good representative of the common case). Then use the new algorithm and shuffle them. We should do this 6 times in a row. This is the result:

From the example above we can see that **the result is correct** – we have randomly shuffled six cards. If our algorithm works well with 6 cards, it should work with decks with different number of cards as well. If we are not sure in that, we should think of another more complicated example and then test the algorithm again.

Otherwise we could avoid drawing new examples and continue with our task.

Let’s summarize **what we have done so far** and how with consecutive actions we have figured out a solution for our problem. As we have gone through every step, we have done so far the following steps:

- We have **used a sheet of paper and pen** to sketch a deck of cards. We have visually represented the deck of cards as an array of boxes.
- As we already have a visual feedback, we could easily think of some sample ideas: firstly we should make **some kind of a single swap operation** and secondly we do this N number of times.
- We had decided that our **"single swap" operation** was going to be splitting the deck at random position into left and right part and then swap them.
- We have decided that we should do this "single swap" as much times as the **number of cards** in the given deck.
- We have considered the problem of **choosing a random number**, but have finally decided to use a ready solution for the job.

- We have **decomposed the main problem** into three smaller subtasks: "single swap" operation; choosing a random split point; combining a sequence of "single swap" operations.
- We have **checked our idea for mistakes** and found one. It was a good thing to check it when we did, because it was not too late to fix it.
- We have thought of a new, **more reliable solution** of the single swap operation.
- We have **checked our new idea with an appropriate example** and we assured ourselves that this time the solution was right.

Now we finally have a working idea, backed up with good examples. This is the most important thing to do in order to solve a given problem – inventing of the algorithm. The easier part remains – **the implementation of our idea**. Let’s see how this can be done.

Choose Appropriate Data Structures!

If we already have a correct and working idea for the solution of the problem, the next thing to do is to write the program code. We have missed something, right? What have we missed? Have we done everything necessary to be able to write fast, easy and trouble-free implementation of our solution?

The thing that we have missed is the manner in which our idea (which we have checked on a sheet of paper) is going to be implemented as a computer program. The **implementation is not always a simple task** and sometimes it requires additional ideas. This is the next major step: to think of our ideas in terms of the computer programming. This means to think for specific data structures and not for abstract ones like "card" and "deck". We should choose the right data structures, which are going to help us build a correct solution.

	Before you even start with the implementation of your idea, you should choose the proper data structures. It may turn out that your current idea is not as good as it seems. The solution could be inefficient or difficult to implement. It is better to figure this out before you write any programming code!

In our case we have spoken of swapping one card from the deck with another, but in terms of programming this means to **swap two elements from specific data structure** (i.e. array, list or something else). We have reached the moment where we have to choose one data structure and show you how it is done.

What Kind of Data Structure Should We Use?

The first question that comes to our mind is: **What kind of data structure should we use?** We may have all kinds of different ideas for data structures, but not all of them can do the work. Let’s reflect for a while, shall we? We

have a collection of cards and the way in which the cards are ordered matters. That's why we need a data structure that can hold a collection of elements and keep their order.

Can We Use an Array?

The first thing we can think of is using **the structure "array"**. The array structure is the simplest data structure, which can hold a collection of elements. The array also keeps the order of the elements (first, second, third and so on) and we can reach each element by index. The array has a fixed number of elements and we cannot change its size during the execution of the program.

Is the **array** the correct data structure for us? To answer this question we have to know what kind of operations we are going to apply on the deck, represented as an array, and whether they are feasible and efficient.

What kind of operations are we going to apply in order to implement our algorithm? Let's enumerate them:

- **Choosing a random card**. Since we can access every element from the array by index we can easily pick a random position k between the interval [1…N-1].
- **Swapping the first card with the k-positioned one** (single swap). After choosing the random card, we should swap it with the first one. Again this operation seems easy enough. We can do the swap with three simple steps and one temporary variable.
- **More operations** that we might use: initialization of the deck; traversing the deck; printing the deck. All these operations seem trivial when applied on array.

It seems that one simple data structure like the array can represent a deck of cards quite well.

Can We Use Another Data Structure?

It is normal to ask ourselves whether an array is the best data structure for our problem. It seems that every operation that we use in our algorithm can be applied efficiently to the array.

But still, let's try and think of an even better data structure for the deck of cards than the array. What other options do we have?

- **Linked list** – we do not have an indexer and it will be difficult for us to access element at a random position.
- **Array with a non-defined size (`List<T>`)** – this structure seems to have all the benefits of the arrays and we can apply every operation to it as well. If we use **`List<T>`**, we increase our comfort – we can easily remove and add elements, which may help us to initialize the deck faster and do some other helpful operations.

- **Stack** / **queue** – the deck of cards does not have a behavior of FIFO or LIFO, so these structures are not appropriate for our algorithm.
- **Sets (`TreeSet<T>` / `HashSet<T>`)** – with the use of sets we lose the original order of the elements which is a major obstacle. The use of sets is inappropriate.
- **Hash table** – the structure card deck is not from the type key-value, so the structure hash table cannot store the deck efficiently. Also it does not allow us to keep the original order of the elements.

Generally speaking, we have just **covered the basic data structures**, which can hold a collection of elements. We have reached the conclusion that either array or **`List<T>`** will be suitable for the job. **`List<T>`** is more flexible than the ordinary array, so we decide to use **`List<T>`** to represent our deck of cards.

	The choice of data structure begins with the consideration of all key operations that we are going to perform on it. Next we analyze all suitable structures and choose the one that will be the most efficient and easiest to use. And sometimes we should make a compromise between efficiency and the simplicity.

How to Represent the Other Data Objects?

We have already decided **how to represent our deck of cards** and now we should do the same with the other objects that we are going to use in our algorithm. If we think about it, it seems that beside the two objects a "**card**" and "**deck**", which we use in our algorithm, we do not use other data objects.

The next question that arises is **how to represent a single card**? We can represent it as a string, number or class, which has two fields – face and suit. There are, of course, other variants, which have their advantages and disadvantages.

Before we even start considering which of these representations of one card is "the best", we should go **back to the requirements** of the task. It suggests that we are given a deck of cards (as an array or list) and our task is to shuffle it. How a card is represented is not of importance in the task. So it does not matter what we shuffle, we could shuffle cards, chess figures, boxes of tomatoes or other objects. We have an **ordered collection of elements and we need to randomly shuffle it**. The fact that we shuffle cards is not significant for our task, that's why we do not need to waste time to choose the best way to represent one card. Let's use the first thing that come to our mind, i.e. we will define a class **`Card`** with 2 fields – **`Face`** and **`Suit`**. Even if we use a number between 1 and 52 to represent one card, it still does not change anything. We shall not discuss this any further.

Sorting Numbers: Choosing a Data Structures

Let’s go back to the **sorting numbers problem** and choose an **appropriate data structures** for it too. We choose to use the simplest algorithm that we could think of: to pick the smallest number until we can, print it and after that delete it. This solution can be easily sketched on a piece of paper and checked for errors.

Again, in order to answer this question we need to figure out **what kind of operations we are going to use in our algorithm**. The operations are as follows:

- **Searching for the smallest number** in the structure.
- **Removing** of the previously found smallest number.

Obviously, the use of an **array** is not reasonable, because we need the operation "remove". The use of **List<T>** seems better, because both operations can be simply and easily implemented. Data structures like **stack** or **queue** have a little use for us, because we do not have a LIFO or FIFO behavior. There is not much sense to use a **hash table**, because the "search by value" operation is not fast, despite the fact that the removal of an element should be very efficient.

Let’s talk about the two **sets** – **HashSet<T>** and **TreeSet<T>**. The two sets have one major problem. They cannot contain elements with an equal value. Despite that let’s see what they can do. The **HashSet<T>** is not of any interest, because like the hash tables it does not support efficient way to find the element with the smallest value. The data structure **TreeSet<T>**, however, looks very promising. Let’s take a look, shall we?

The **TreeSet<T>** class is a **balanced search tree** by design, so it supports the operation "finding the smallest element". That’s interesting, isn’t it? Now we have a new solution for the task, we put all the input elements in a **TreeSet<T>** and then we get the smallest from the set until it remains empty. Easy, simple and very efficient. The two operations, which we want, are internally implemented (searching for the smallest number and deleting it).

While we skim through the documentation, we figure out something very interesting: the **TreeSet<T>** stores its **elements ordered by value**. And this is the solution of our problem, right? Therefore if we keep all the input elements in a **TreeSet<T>** and then traverse the ordered set (with the help of the built-in enumeration), we will have all the elements ordered by value. Problem solved!

We are now very happy, we found one very nice way to solve our task, but soon we discover one major problem: **TreeSet<T>** does not store two **elements with the same value**. I.e. if we add the number 5 several times, at the end there will be only one entry with a value 5. Eventually we will lose some of the input elements irreversibly.

Naturally we want this problem fixed. If there was a way to store how many times one number occurs in a set that would solve our problem. Then we think of the **SortedDictionary<K,T>**. This class can store ordered keys, which have a value. We can store the **number of occurrences** of a key in its corresponding value. We can traverse all the elements and then store the number of occurrences in the **SortedDictionary<K,T>**. Although it seems our problem is solved, it is not going to be implemented as elegant and simple as with **List<T>** or **TreeSet<T>**.

If we read the documentation of the **SortedDictionary<K,T>** carefully, we will find that this class internally uses a **red–black tree** and some day we can implement that this type of sorting is very famous and it is called a Binary Tree Sorting (http://en.wikipedia.org/wiki/Binary_tree_sort).

With this little demonstration we showed you how when you put some thoughts into the **selection of the best data structures**, you can come up with some new solutions for the problem. We start with an algorithm, which leads us to a new, better one. This is normal to happen during the process of consideration of our algorithm and not after we have written 300 lines of code, which we will then have to be redone. This is another proof it is better to firstly think of the best data structures and then to start writing the programming code.

Think about the Efficiency!

Again, it seems we should grab the keyboard and start writing a programming code. And again, **it is better not to hurry**. The thing is that we have not thought of something very important: the **efficiency and performance** of our algorithm.

You should think of efficiency before writing even a line of a programming code. Otherwise, you risk to waste time implementing an algorithm, which is inefficient and slow.

Let's return to our "card-shuffle" problem. We have a working idea for solving the problem (we have invented the algorithm). The idea appears to be correct (we have checked the algorithm with examples). We should not have any problems implementing our idea (we are going to use **List<Card>** for the deck and class **Card** for a single card). Everything seems fine, but let's think about **how many cards we are going to shuffle**. Is our idea going to work fast enough when using the chosen data structures?

How to Estimate the Performance of Given Algorithm?

How fast is our algorithm? To answer this question we should estimate **how many operations it performs** when shuffling one deck of 52 cards.

For one deck of 52 cards our algorithm makes **52 "single swap" operations**, do you agree? How many elementary operations cost one "single

swap"? 4 operations: the choice of one random card; the placing of the first card in a temporary variable; the replacing of the first card with the random card; the replacing of the random card with the first card (from the temporary variable). How many operations does our algorithm do? They are approximately 52 * 4 = 208.

Are 208 operations too much? Let's do a loop with 208 iterations. Are they too much? Give it a try! We can assure you that one loop with 1,000,000 iterations on a modern computer goes imperceptibly fast, and one with 208 – for an insignificant amount of time. Therefore we can easily conclude that **our algorithm has a good performance**. Our algorithm is extremely fast when working with 52 cards.

Despite the fact that in reality we rarely play cards with more than 1 – 2 decks, let's assume that **we have 50,000 cards in the deck**. Let's estimate the performance of our algorithm with a large number of cards. We have 50,000 single swap operations and each of them consists of 4 operations, which makes about 200,000 operations, which are going to be executed for a small amount of time as well.

The Efficiency Is a Matter of Compromise

Finally we can conclude that our algorithm is efficient and will work well even with decks with large amount of cards. Here we had luck. Usually the things are not so simple and we must make a compromise with the performance and the efforts, which we put, when we implement our algorithm. For example if we sort numbers, we can solve this problem in minutes when we use some of the simplest sorting algorithms. We can also do this much more efficiently when we use some of the **more complex algorithms, but that will waste more of our time** (in searching and reading books and Internet).

Is it worth it? We should consider that. If we have to sort 20 numbers, it does not matter which algorithm we are going to use. It will always be fast, even with the most naive algorithm. If we are going to sort 20,000 numbers, the algorithm matters, and if we need to sort 20,000,000, we should **look at the task from a completely new angle**. The efforts for solving efficiently the problem of sorting 20,000,000 numbers is far more than the efforts for writing a straightforward algorithm to sort 20 numbers. We should answer the question**: is it worth it**?

The efficiency is a matter of compromise – sometimes it does not worth to complicate your algorithm and put time and effort to make it work faster. But occasionally the performance is crucial and we should pay serious attention to it.

Sorting Numbers: Estimating the Performance

It is obvious that the performance depends on whether a particular task requires it. And now let's return to the sorting numbers problem, because we want to show you that the **efficiency is directly related to the right choice of data structures**.

Let's go back to the point where we have decided what kind of data structures to use for keeping the input data. Which is better: **List<T>** or **SortedDictionary<K,T>**? Shouldn't we use a data structure that we know well instead of some complex structure that we have never used? Do you know well red-black trees (the internal implementation of the **SortedDictionary<K,T>**)? With what are they better than **List<T>**? In fact it may turn out that you do not need to answer this question after all.

If we have to **sort 20 numbers**, does it matter what data structure are we going to use? **We can choose the simplest algorithm** and the first data structure that is actually suitable for the job and that's it. It does not matter how fast is the algorithm and the data structure, because the numbers are not so many.

But if we have to **sort 300,000 numbers**, then everything is different. We should carefully study how exactly the class **SortedDictionary<K,T>** behaves. We should figure out **how fast** is the "search" operation. How fast does this data structure add elements? How fast can you traverse through every element of the collection? If we read the documentation of the class we will see that the adding of an element takes on average $\log_2(N)$ steps, where N is the number of the elements in the structure. After few simple mathematical calculations (which require additional skills), we can roughly estimate that we need about 5-6 million steps to sort all numbers. For 300,000 numbers this number is **reasonably small**.

Similarly we can prove that the search and delete operations in **List<T>** with N elements take N steps. Therefore for 300,000 elements we will need roughly 2 * 300,000 * 300,000 steps. In fact this number is an approximate guess, because at the beginning we have one number in the list, not 300,000 elements. Nevertheless this estimation is approximately right, maybe a bit rough but right. We can see that **the number of steps** needed in this case **is extremely large**, that is why here the simple **algorithm will not work** properly (the program might "hang").

And again we reach a point where we need to **choose between one simple and one complex algorithm**. One of them can be very easily but slow when implemented. The other is more efficient, but very difficult to implement and we will probably need an additional reading of documentation and thick books in order to correctly estimate the performance. **Everything is a matter of compromise**.

Naturally, at this point we can think of some of the other algorithms that we have considered previously. And precisely, to **split the array into two parts** then to **sort them separately** (by a recursive call) and then merge the two

parts into one sorted array. As we consider this algorithm we will find that this solution will work efficiently with such structures like the dynamic array (**List<T>**). This sorting algorithm has an **average and worst-case performance of n*log(n) steps**, where n is the count of the elements in the array. This algorithm will work efficiently with 300,000 numbers. Let's not go any further, if you want more details about the algorithm you should read more about **MergeSort** in Wikipedia (http://en.wikipedia.org/wiki/Merge_sort).

Implement Your Algorithm!

We have finally reached the time where we can start with the **implementation of our solution**. We already have a working idea, we have chosen the best data structure and now it is the time to start writing the programming code. If we have not done some of the previous steps, we should go back to them before start writing the code.

If you do not have an invented idea, do not start writing programming code! What are you going to write if you do not have a working idea? This is like to go to the train station and get on the first train that you can see, without even deciding where you are going.

This is **typical for novice programmers**: once they see the requirements, they proceed with the writing of the programming code. After some time, that they waste in a pursuit of wrong ideas (that occur to them during the writing), they realize that it is better to **stop and think a bit** more about the solution. This whole concept is wrong and the main goal of this chapter is to protect you from this frivolous and very inefficient approach to problem-solving.

If you have not checked your ideas, there is no sense to start implementing them! Is it necessary to write 300 lines of code before implementing that your idea is totally wrong? Is it necessary for you to start over?

The implementation of already invented and checked idea is very easy and simple. But the implementation itself requires additional skills and mostly **experience**. The more experience you have the faster and easier it will be for you to write efficient programming code. With lots of practice, which will come with time, you will become very skilled in writing high-quality code and you will be able to write code faster. If you want to know more about high-quality programming code you should read the chapter "High-Quality Programming Code". But for now let's focus on the implementation of our ideas.

We assume that you should already know the basic steps needed to write programming code: you know how to work with the development environment (Visual Studio), the compiler; how to understand the error messages and use the "auto complete" function; how to create methods, constructors and properties and fix errors and use the debugger. Therefore these next advices

are not so much connected with the writing itself but with the overall approach when writing programming code.

Write the Code Step by Step!

Have you written 200-300 lines of code without even compiling or testing it? Do not do that! Do not write large lumps of code at one time, instead you should write small parts and then test them.

How to write code step by step? This depends on the given task and the way, in which it is decomposed into smaller tasks. For example if the main task consists of 3 independent parts, we should **write one of them, compile and test it** with a proper input data until we are sure that it works correctly. After that we **move to the second part** – write code, compile, test and then proceed with the **third part** with the same approach and finally **integrate the parts and test everything as a whole**.

Why to write code step by step? Because we **reduce the amount of code that we have to concentrate on in any given moment**. By treating the problem in parts, we decrease its complexity. Remember: the large and complicated task **could always be divided** into several smaller and simpler subtasks. And it is always easier to solve simple problems.

When writing large chunks of code, without compiling it, we **accumulate a great amount of errors**, which could easily be avoided by a simple compilation. The modern programming environments (like Visual Studio) try to recognize the syntactic errors automatically while we are writing the code. Use this function and fix the obvious coding errors as early as possible. **Early troubleshooting takes less time and nerves**. However if we delay the troubleshooting, it could cost us a lot of efforts, sometimes even rewriting the whole programming code.

When you write a huge amount of code, which is not tested, and decide to test it as a whole with some input data, you usually receive a lot of errors, which **can be avoided if one just compiles**. The larger the code is, more difficult it is to be fixed. These problems could be caused by a variety of reasons: incorrect use of data structures; wrong algorithm; badly structured code; bad condition in the `if`-statement; wrongly implemented loop; going out of bounds of the array and many other problems that could have been fixed earlier. **Do not wait for the last moment. Eliminate the mistakes as soon as possible!**

	Write your program in parts, not at once! Take, write and compile one logically independent part, fix the errors, test it and if it works fine, move to the next part.

Writing Code Step by Step – Example

In order to **demonstrate how to write code step by step**, we should illustrate it with the "card-shuffle" algorithm that we invented previously.

Step 1 – Defining the Class "Card"

Our task is to shuffle the card deck, so let's **start with the definition of the class "card"**. If we do not have an idea of how to represent one single card, we could not have any idea how to represent a deck as well. Therefore it will not be possible to define a method for shuffling the cards. We have already agreed the representation of one card does not matter, so any kind of them might work.

We will define a **class "card" with fields face and suit**. We will use a string variable for the face of the card (with possible values: "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" or "A") and enumerable variable for the suit of the card (possible values: "Club", "Diamond", "Heart", "Spade"). The class Card might look like the following code:

Card.cs

```
class Card
{
   public string Face { get; set; }
   public Suit Suit { get; set; }

   public override string ToString()
   {
      string card = "(" + this.Face + " " + this.Suit + ")";
      return card;
   }
}

enum Suit
{
   CLUB, DIAMOND, HEART, SPADE
}
```

For comfort we have overridden the method **ToString()** for the class **Card**. In this way we could easily print a single card on the console. We have defined enumerable type for the **Suit**.

Testing of the Class "Card"

Some of us would probably proceed with writing the code, but if we follow the principle "Writing Code Step by Step", we should **firstly compile and test how the class Card works**.

In order to do so, we can write a small simple program to initialize a single card (e.g. Ace of Clubs) and print it on the console. This will check whether our class **Card**, its constructor and its **ToString()** method work correctly:

```
static void Main()
{
    Card card = new Card() { Face="A", Suit=Suit.CLUB };
    Console.WriteLine(card);
}
```

We start the program and check if the card is printed correctly. We should see the following:

```
(A CLUB)
```

Step 2 – Creating and Printing a Deck of Cards

Before we proceed with the main task (randomly shuffling the deck of cards) we should try to **initialize and print a whole deck of 52 cards**. Thus we will be completely sure that the input data for the card-shuffle method is correct. Based on our previous analysis on the data structures, we should use **List<Card>** in order to represent the deck. Let’s create and print a **deck of five cards**, shall we? Later we can try with a full deck of 52 cards.

CardsShuffle.cs

```
class CardsShuffle
{
    static void Main()
    {
        List<Card> cards = new List<Card>();
        cards.Add(new Card() { Face = "7", Suit = Suit.HEART });
        cards.Add(new Card() { Face = "A", Suit = Suit.SPADE });
        cards.Add(new Card() { Face = "10", Suit = Suit.DIAMOND });
        cards.Add(new Card() { Face = "2", Suit = Suit.CLUB });
        cards.Add(new Card() { Face = "6", Suit = Suit.DIAMOND });
        cards.Add(new Card() { Face = "J", Suit = Suit.CLUB });
        PrintCards(cards);
    }

    static void PrintCards(List<Card> cards)
    {
        foreach (Card card in cards)
        {
            Console.Write(card);
        }
        Console.WriteLine();
    }
}
```

Printing the Deck – Testing the Code

Before we proceed forward, let's start the program and verify the output result. It seems that there are no mistakes, **the result is correct**:

```
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
```

Step 3 – Single Swap

Let's implement the next step to the task solution – the **subtask "single swap"**. When we have a logically independent piece of code, the best thing to do is to extract it as a separate method. We should think of what is our input and output. Our input should be a single deck of cards (**`List<Card>`**). As a result of its work the method should change the input deck **`List<card>`.** The method should not return any result, because it does not create a new **`List<Card>`**, it just operates with the already created and submitted list.

What should be the **name** of the method? Following the recommendations for working with methods, we should give "descriptive" name (with 1-2 words) what the method is for. Suitable name for it is: **`PerformSingleSwap(…)`**. The name clearly describes what the method does: executes a single swap.

Let's firstly define the method and then write its body. This is a good practice, because before we proceed with the implementation of the method, we should be aware: what does it do; how does it work and what is its name. Here it is **the definition of the method**:

```
static void PerformSingleSwap(List<Card> cards)
{
   // TODO: Implement the method body
}
```

The next thing to do is to write the body itself. Firstly let's recall the algorithm: we **choose one random number k** in the interval between 1 and the length of the array minus 1 and then **swap the element at the position k with the first element**. Everything seems easy, but how do we generate a random number in a given interval with the language C#?

Search in Google!

When we encounter a common problem, which we cannot solve, but we are sure that many people have faced it, the easiest way to cope with it is to search for information in Google. We should adequately structure our search. In our case we look for **sample C# code**, which returns as a result a **random number in a given interval**. We could try the following search:

```
C# random number example
```

Among the first results there is a C# program, which uses the class **System.Random** for **generating a random number**. Now we have a direction in which we look for a solution. We know that in .NET Framework there is a standard class called **Random**, which serves for generating random numbers.

After that we could try to guess how this class works (most of the times it takes less time to guess instead of reading the documentation). We are trying to find an appropriate static method for generating a random number, but it seems there is none. Then we make an instance and search for a method, which could return a number in given a diapason. We have luck, there is a method **Next(minValue, maxValue)**, which returns what we need.

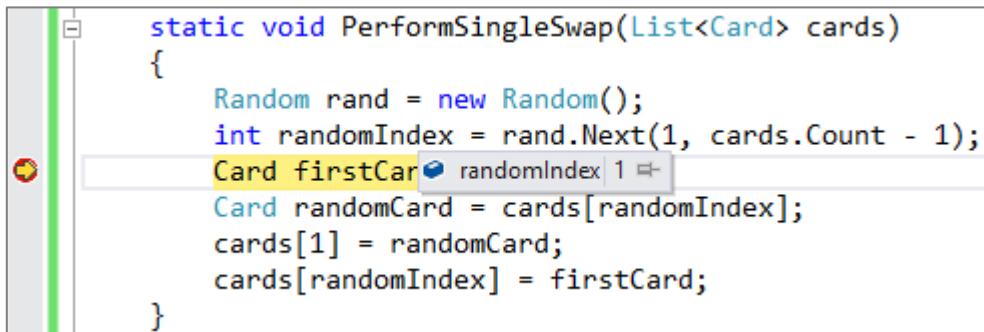
Let's try to write the whole code for the method. We have the following:

```
static void PerformSingleSwap(List<Card> cards)
{
  Random rand = new Random();
  int randomIndex = rand.Next(1, cards.Count - 1);
  Card firstCard = cards[1];
  Card randomCard = cards[randomIndex];
  cards[1] = randomCard;
  cards[randomIndex] = firstCard;
}
```

Single Swap – Testing the Code

The next step is to **test the code**. Before proceeding forward, we have to be sure that the single swap (exchange) operation works properly. We do not want to find an eventual problem just when we test the "card-shuffle" method with the entire deck? It is better when there is a **problem to be found immediately** and when there is none, to continue forward with confidence. We act step by step – before going to the next step we should make sure that the current step is working fine. For this purpose we make a small test program, let's say with 3 cards (2♣, 3♥ and 4♠):

```
static void Main()
{
  List<Card> cards = new List<Card>();
  cards.Add(new Card() { Face = "2", Suit = Suit.CLUB });
  cards.Add(new Card() { Face = "3", Suit = Suit.HEART });
  cards.Add(new Card() { Face = "4", Suit = Suit.SPADE });
  PerformSingleSwap(cards);
  PrintCards(cards);
}
```

Let's perform **several times a single swap operation with our 3 cards**. The first card (card 2♣) is supposed to be with one of the other two cards (cards 3♥ or 4♠). We execute the program several times in a sequence. We

should expect the half of the obtained results to contain (3♥, 2♣, 4♠) and the other half – (4♠, 3♥, 2♣), shouldn't we? Let's see what is going to happen. We start the program and see the following results:

```
(2 CLUB)(3 HEART)(4 SPADE)
```

We start it again and again and the result is the same – **no swap is made**. How is that possible? What has just happened? Did we miss to execute the single swap before printing the cards? There is something wrong here. It seems that the program did not make even one swap in the deck of cards. How did this happen?

Single Swap – Correcting the Mistakes

It is obvious that there is a **mistake**. Let's put a **breakpoint** and follow what is happening via the debugger of Visual Studio:

```
static void PerformSingleSwap(List<Card> cards)
{
    Random rand = new Random();
    int randomIndex = rand.Next(1, cards.Count - 1);
    Card firstCar randomIndex 1
    Card randomCard = cards[randomIndex];
    cards[1] = randomCard;
    cards[randomIndex] = firstCard;
}
```

It is clear that during the first execution the random position happens to be one. This is acceptable so we continue on. When we look the code we follow, we notice that **we swap the random element** at index 1 with the element at position 1 i.e. **with itself**. We apparently **did something wrong**. And then we remember that indexing in **List<T> is zero-based** i.e. the first element is at position 0. We immediately change the code:

```
static void PerformSingleSwap(List<Card> cards)
{
  Random rand = new Random();
  int randomIndex = rand.Next(1, cards.Count - 1);
  Card firstCard = cards[0];
  Card randomCard = cards[randomIndex];
  cards[0] = randomCard;
  cards[randomIndex] = firstCard;
}
```

We start the program several times and we get **unexpected results**, again:

```
(3 HEART)(2 CLUB)(4 SPADE)
```

```
(3 HEART)(2 CLUB)(4 SPADE)
(3 HEART)(2 CLUB)(4 SPADE)
```

It seems that **the random number is not so random**. What to do now? Do not rush to blame .NET Framework, CLR, Visual Studio and all other usual suspects! It is possible that **the mistake is ours**. Let’s look at the execution of the method **Next(…)**. Since cards' count is 3, we always call **Next(1, 2)** and expect from it to return a number between one and two. It seems correct but if we read what the documentation says for the method **Next(…)**, we will notice that the second parameter should be one unit bigger than the upper border we want to obtain.

We were **wrong about the diapason of the random number** that we selected. We correct the code and once again we test it to see how it works. After the second correction we get the following results:

```
static void PerformSingleSwap(List<Card> cards)
{
   Random rand = new Random();
   int randomIndex = rand.Next(1, cards.Count);
   Card firstCard = cards[0];
   Card randomCard = cards[randomIndex];
   cards[0] = randomCard;
   cards[randomIndex] = firstCard;
}
```

Here are the possible results after several executions of the previous method:

```
(3 HEART)(2 CLUB)(4 SPADE)
(4 SPADE)(3 HEART)(2 CLUB)
(4 SPADE)(3 HEART)(2 CLUB)
(3 HEART)(2 CLUB)(4 SPADE)
(4 SPADE)(3 HEART)(2 CLUB)
(3 HEART)(2 CLUB)(4 SPADE)
```

It seems that after enough executions the first card is replaced by each of the other two cards i.e. **we have a random swap** indeed and every card has the equal chance to be randomly chosen. We are finally ready with the method "single swap". It is better that we **found these two mistakes now and not later** when the whole program is supposed to start working, right?

Step 4 – Card Shuffling

The last step is simple: we use the **single-swap method N times**:

```
static void ShuffleCards(List<Card> cards)
{
```

```
    for (int i = 1; i <= cards.Count; i++)
    {
        PerformSingleSwap(cards);
    }
}
```

We now can put it all together. We **combine all the pieces of code** we already wrote, tested and we checked they work correctly. The entire code of our program looks like this:

CardsShuffle.cs

```
using System;
using System.Collections.Generic;

class CardsShuffle
{
    static void Main()
    {
        List<Card> cards = new List<Card>();
        cards.Add(new Card() { Face = "2", Suit = Suit.CLUB });
        cards.Add(new Card() { Face = "6", Suit = Suit.DIAMOND });
        cards.Add(new Card() { Face = "7", Suit = Suit.HEART });
        cards.Add(new Card() { Face = "A", Suit = Suit.SPADE });
        cards.Add(new Card() { Face = "J", Suit = Suit.CLUB });
        cards.Add(new Card() { Face = "10", Suit = Suit.DIAMOND });

        Console.Write("Initial deck: ");
        PrintCards(cards);

        ShuffleCards(cards);
        Console.Write("After shuffle: ");
        PrintCards(cards);
    }

    static void PerformSingleSwap(List<Card> cards)
    {
        Random rand = new Random();
        int randomIndex = rand.Next(1, cards.Count);
        Card firstCard = cards[0];
        Card randomCard = cards[randomIndex];
        cards[0] = randomCard;
        cards[randomIndex] = firstCard;
    }
```

```
    static void ShuffleCards(List<Card> cards)
    {
        for (int i = 1; i <= cards.Count; i++)
        {
            PerformSingleSwap(cards);
        }
    }

    static void PrintCards(List<Card> cards)
    {
        foreach (Card card in cards)
        {
            Console.Write(card);
        }
        Console.WriteLine();
    }
}
```

Card Shuffling – Testing

Now we only need to **test** whether the algorithm for shuffling a deck of cards works correctly. Here is the output of our program:

```
Initial deck: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
After shuffle: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
```

As we can see, we encounter a problem: after the shuffle the cards did not change their order. We run the program several times and the result is the same. Did we forget to call the card-shuffling method **ShuffleCards**?

We take a close look at our source code: **everything looks fine**. We decide to set a breakpoint after the call of the method **PerformSingleSwap(…)** in the body of the loop, responsible for the shuffling of the cards. We **run our program in debugging mode** by pressing the **[F5]** button. After the first stop of the debugger at our breakpoint **everything seems good** – the first card is exchanged with another one, as it supposed to. After the second stop of the debugger everything is **still all right** – a random card is swapped with the first one. We continue tracing the program execution **with the debugger** and everything seems to work just fine.

The card shuffling program **works flawlessly** when we run it step by step through the Visual Studio debugger:

```
static void ShuffleCards(List<Card> cards)
{
    for (int i = 1; i <= cards.Coun
    {
        PerformSingleSwap(cards);
    }
}

static void PrintCards(List<Card> cards)
```

```
cards | Count = 6
[0]       {(10 DIAMOND)}
[1]       {(6 DIAMOND)}
[2]       {(J CLUB)}
[3]       {(2 CLUB)}
[4]       {(7 HEART)}
[5]       {(A SPADE)}
Raw View
```

But why is the final result wrong? We decide to set another breakpoint in the body of **ShuffleCards(…)** at the end. The debugger stops and at this point and the result is still correct – the cards are randomly shuffled. We continue debugging and we reach the place where we print the deck. We go pass it and the cards are printed to the console in random order. **Strangely: still the correct result. What is the problem?**

We start the program **without debugging** it with **[Ctrl+F5]**. The result is **wrong** – the cards are not shuffled. We run our program in debugging mode again with the press of **[F5]**. The debugger once more stops at the breakpoints and the program yet again is working without any problem. It looks like that, when we run our program **in debug mode the result is correct**, but when we start it normally, **without the debugger, the answer is wrong**. Strange indeed!

We decide to add a line of code, which is going to **print the deck after every single swap**:

```
static void ShuffleCards(List<Card> cards)
{
   for (int i = 1; i <= cards.Count; i++)
   {
      PerformSingleSwap(cards);
      PrintCards(cards);
   }
}
```

We run our program in debug mode (with **[F5]**), observe the execution step by step and we find that it **works correctly**:

```
Initial deck: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
(A SPADE)(7 HEART)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
(6 DIAMOND)(7 HEART)(10 DIAMOND)(2 CLUB)(A SPADE)(J CLUB)
(J CLUB)(7 HEART)(10 DIAMOND)(2 CLUB)(A SPADE)(6 DIAMOND)
(2 CLUB)(7 HEART)(10 DIAMOND)(J CLUB)(A SPADE)(6 DIAMOND)
```

```
(A SPADE)(7 HEART)(10 DIAMOND)(J CLUB)(2 CLUB)(6 DIAMOND)
(10 DIAMOND)(7 HEART)(A SPADE)(J CLUB)(2 CLUB)(6 DIAMOND)
After shuffle: (10 DIAMOND)(7 HEART)(A SPADE)(J CLUB)(2 CLUB)(6
DIAMOND)
```

We run again our program in normal mode (with **[Ctrl+F5]**) and the answer is **still incorrect**. Yet again we try to find out why it is happening:

```
Initial deck: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
(6 DIAMOND)(A SPADE)(10 DIAMOND)(2 CLUB)(7 HEART)(J CLUB)
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
(6 DIAMOND)(A SPADE)(10 DIAMOND)(2 CLUB)(7 HEART)(J CLUB)
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
(6 DIAMOND)(A SPADE)(10 DIAMOND)(2 CLUB)(7 HEART)(J CLUB)
(7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6 DIAMOND)(J CLUB)
After shuffle: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
```

We can clearly see that at every step, on which we expect to be done a single-swap, actually **the same cards are swapped: 7♥ and 6♦**. The only way this to happen is if every time **the random number is the same**. The conclusion is that we have a problem with the generation of random numbers. **The random generator does not work correctly**.

We instantly think about taking a look at the documentation of the class **System.Random()**. On MSDN we can read, that by creating a new instance of the generator of pseudo-random numbers with the constructor **Random()**, the generator is initialized with a value, equal to **the current system time**. In the documentation we can further read that by creating two or more instances of the class **Random** in a relatively short time span, there is a great chance the numbers to be the same. It turns that the problem consists in the misuse of the class **Random**.

Now being more familiar with the current problem, we could correct it by **creating an instance of the class Random only once** at the beginning of the program. After that, if we need a random number, we are going to use the already created generator of pseudo-random numbers. After the correction, the code looks like this:

```
class CardsShuffle
{
  …

  static Random rand = new Random();

```

```
    static void PerformSingleSwap(List<Card> cards)
    {
        int randomIndex = rand.Next(1, cards.Count);
        Card firstCard = cards[0];
        Card randomCard = cards[randomIndex];
        cards[0] = randomCard;
        cards[randomIndex] = firstCard;
    }
    …
}
```

It seems that **the program finally works correctly**. At every run the order of the cards is different and looks randomly:

```
Initial deck: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
After shuffle: (2 CLUB)(A SPADE)(J CLUB)(10 DIAMOND)(7 HEART)(6
DIAMOND)
--------------------------------------------------------
Initial deck: (7 HEART)(A SPADE)(10 DIAMOND)(2 CLUB)(6
DIAMOND)(J CLUB)
After shuffle: (6 DIAMOND)(10 DIAMOND)(J CLUB)(2 CLUB)(A
SPADE)(7 HEART)
```

We try some other tests and the final result is still **correct.** Now we can say that we have a correct implementation of our algorithm, which we have designed earlier.

Step 5 – Console Input

Now we only need to be able to **read the deck from the console** so that the user can enter the cards, which need to be shuffled. Notice that we **left this step as last**. Why? The answer is pretty simple: we do not want to enter the same data at the start of our program just to check whether a piece of our code works correctly. By having the needed data hard-coded in the source code, we save a lot of time during the developing process.

If the problem involves entering data from the console, leave this as last step and be sure that everything else works flawlessly before implementing the reading of the input. While writing the source code, do your tests with hard-coded examples so that you don't have to enter any data. This way you are going to save your time and your nerves.

Data entry is a low-priority task, which everyone can handle. You only need to think of the format the cards are entered – are they entered one by

one or at once; are the face and the color entered at once or separately. There is nothing difficult so we leave this to our readers.

Sorting Numbers – Step by Step

Till this moment, we showed you **how important it is to write your program step by step** and before proceeding to the next step you must be sure that the previous one is working correctly.

Let's take a look at the problem of **ordering numbers in ascending order**. The steps for solving this problem are the same. Once again the right approach for solving this task is to **work step by step**. Let's see shortly which the steps are. We are not going to write any code, but we are going to consider the main parts of the solution. Suppose that we use **`List<int>`**, in which we successively find the lowest number, print it and erase it from the list of numbers. These are the steps:

> **Step 1.** Think up a good **test data** (example). For example the numbers: 7, 2, 4, 1, 8 and 2. We create a **`List<int>`** and fill it with the numbers from the example. We create a method, which outputs the numbers.
>
> We compile and **test the piece of code we just wrote**.
>
> **Step 2.** We create a method to **find the lowest number** in the array and return its position.
>
> We **test the method**, responsible for the search of the lowest number. We try different sets of numbers to be sure that the search works correctly (we set the lowest element at the beginning, at the end, at the middle; we also consider a case when the lowest number occurs more than once).
>
> **Step 3.** We create a method to **find the lowest number**, print it and after that delete it.
>
> We **test with our example** if the method works correctly. We must also try with other examples.
>
> **Step 4.** We create a method to **sort the numbers**. This method executes N times the previous one (where N is the count of the numbers).
>
> We must **test** whether everything works correctly.
>
> **Step 5.** If data input is required we implement it after everything else.

You can see that the approach to **break the main problem into smaller problems** can work well for all tasks. Simply we just need to figure out which are the smaller problems, from the bigger one, and implement them. On every smaller problem implemented we need to test it. After every step we need to start our program to be sure that till this moment everything works flawlessly. In this way we are be able to find out errors quickly and debug them. It would be much faster than trying to debug them after we have written tons of code.

Test Your Solution!

Does the following sound familiar to you: "I am ready with the first task so I have to start the next one."? Everybody has thought of this on an exam. But in programming **to be ready with a task means**:

1. I have **understood** well the description of the task.
2. I have come up with an **algorithm** to solve the problem.
3. I have **tested my algorithm** on a piece of paper and I am sure that is correct.
4. I have thought up for the **data structures** we need and for the complexity of my algorithm.
5. I have **written a program**, which implements my algorithm correctly.
6. I have **tested my program** with suitable examples so that I can be sure that everything works flawlessly, even in unusual situations.

Inexperienced software developers often **forget about the last step**. They think that testing is not their job, which is their biggest mistake. It is like to think that Microsoft is not supposed to test Windows and let it crash every minute.

Testing is an important part of the programmer's duties. Writing code without testing is like typing on the keyboard without looking at the screen – you think that the text, you have written, is correct, but most likely it is full of bugs.

Experienced programmers know that **untested code is not finished**. In most software companies it is completely unacceptable not to test your work.

In the software industry is widely spread the idea of **unit testing** – **automatic testing of every unit of code** (methods, classes and modules). Unit testing means, that for every program, we create another one, which to test our work. In some companies firstly they think up the testing scenario, build the tests and only then start working on the program. The things you should know about unit testing are quite many, but you will get more familiar with them as you get deeper in the "software engineering". For now we are going to focus on the manual testing every programmer must do. Unit testing frameworks and test automation can be used to simplify the process.

How to Test?

A program works correctly if it can handle every kind of input data. **Testing** is a process, which aims to **find any type of bugs**. It cannot detect whether a program works flawlessly, but it can help you to find most of the bugs, which cause incorrect results and other types of errors.

Sadly you can't predict all cases and test them. Therefore you must think up examples, which cover most of the situations, which could happen. In this

way you could with minimum efforts (i.e. with minimum count of simple tests) to **check all common cases** of usage of the program. If no bugs are found after testing, this doesn't mean that the program works 100% correctly, but in this way we reduce the chance of the program to crash in a later phase.

Testing can only find the existence of bugs. It can't prove that a program works flawlessly! Programs, which are carefully tested, have fewer bugs than untested or not carefully tested programs.

It is good to **start testing with a typical case** for our program. Often this is the same example we have tested on a piece of paper and which our algorithm can handle correctly.

Normally, after the code is written we only need to fix some minor bugs so that our program can pass the test correctly. After that we have to test our program with **more difficult and bigger examples** to see how our program behaves in more complicated situations. We now have to test with **borderline cases** and test for **performance**. Depending on the complexity of the current task, we do from 1-2 to dozens of tests to cover the main cases of usage.

With more complicated software, i.e. Microsoft Word, the number of tests can be thousands, even **hundreds of thousands**. If a program's functionality is not carefully tested you can't say that it works correctly. Testing during software development is as **important as writing the code**. In big software companies for every programmer there is a tester. For example in **Microsoft** for every programmer, who writes the code (software developer), there are **two people hired to test the code** (software quality assurance engineers). Although these people do not write the main software, they write testing software and therefore we call them software engineers.

Testing with Good Examples of the Common Cases

As we already mentioned, it is good to start testing with a good example of a common situation. This is a test, which is **enough simple to be written down** on a piece of paper and **accurate enough to cover the usual usage** of the program excluding special cases. The steps are as follows:

1. Think up a test, which is a good example of a common situation.
2. Test this example on a piece of paper.
3. Expect the program to work correctly for that test.
4. Be sure that the example works correctly after the program is written and the errors in the development process are fixed.

Sadly, many programmers **end their testing at this moment**. Some inexperienced programmers do something even worse: think up a stupid test case (which is a special case for the current program), don't write it down, write some code and, after the program passes the example, they continue. **Don't do like this!** This is like repairing a car and once you are finished,

driving it downhill, thinking that the car is repaired (but the car has no power to drive uphill).

What Else to Test For?

Testing the case, drawn on a piece of paper, is **only the first step**. Next you need to do some additional tests to be sure that your program works correctly:

- A **hard common-case test**. The goal is to check whether your program can handle a bigger and harder to compute example. For our task that kind of a test is to shuffle a deck of 52 cards.
- **Borderline tests**. They check whether your program can handle an unusual case, which could happen. In our case this could be shuffling a deck, which contains only one card.
- **Performance tests**. These tests put our program in extreme conditions. Usually these tests consist of large data, which needs to be inputted and processed.

Let's take a look at the groups of tests above one be one.

A Hard Common-Case Test

We have already **tested our program for one case**, which we have written down on a piece of paper. Our program works correctly. This case covers a typical scenario of usage of our program. What more do we have to test for? It is possible our program to be incorrect, but accidentally to work for our test.

How to think up a **harder test**? It depends on the task. It must consist of larger amounts of data and we must be able to see whether the output of our program is correct.

In our case we have to test with a full deck – 52 cards. We can easily produce such a test with two nested loops. After the execution of our program we could also easily **check whether the answer is correct** – the cards must be randomly shuffled. It is necessary to check whether the cards are again randomly shuffled after two consecutive executions of this test. The code for this test looks like this:

```
static void TestShuffle52Cards()
{
  List<Card> cards = new List<Card>();
  string[] allFaces = new string[] { "2", "3", "4", "5",
    "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
  Suit[] allSuits = new Suit[] { Suit.CLUB, Suit.DIAMOND,
    Suit.HEART, Suit.SPADE };
  foreach (string face in allFaces)
```

```
    {
        foreach (Suit suit in allSuits)
        {
            Card card = new Card() { Face = face, Suit = suit };
            cards.Add(card);
        }
    }
    ShuffleCards(cards);
    PrintCards(cards);
}
```

If we execute it, the returned result is:

```
(4 DIAMOND)(2 DIAMOND)(6 HEART)(2 SPADE)(A SPADE)(7 SPADE)(3
DIAMOND)(3 SPADE)(4 SPADE)(4 HEART)(6 CLUB)(K HEART)(5 CLUB)(5
DIAMOND)(5 HEART)(A HEART)(9 CLUB)(10 CLUB)(A CLUB)(6 SPADE)(7
CLUB)(7 DIAMOND)(3 CLUB)(9 HEART)(8 CLUB)(3 HEART)(9 SPADE)(4
CLUB)(8 HEART)(9 DIAMOND)(5 SPADE)(8 DIAMOND)(J HEART)(10
DIAMOND)(10 HEART)(10 SPADE)(Q HEART)(2 CLUB)(J CLUB)(J SPADE)(Q
CLUB)(7 HEART)(2 HEART)(Q SPADE)(K CLUB)(J DIAMOND)(6 DIAMOND)(K
SPADE)(8 SPADE)(A DIAMOND)(Q DIAMOND)(K DIAMOND)
```

If we take a careful look we notice that most of the cards are at the same place. After the first 4 cards, the **half of the cards didn't change their place** after the shuffle: 2♦ and 2♠.

It is never late to find a bug in our program and the only way we can do that is to test it with many tests, which cover practical situations. It was useful to test with a real deck of 52 cards, wasn't it? We came upon a serious bug, which cannot be left unfixed.

How to fix the problem? The first idea, we come up with, is to do more single-swaps (obviously N times are not enough). Another idea is at the N^{th} single-swap to **swap the N^{th} card from the deck with a random one instead of changing it with the first one**. In this way we can guarantee that every card is going to be swapped. The second idea is better. Let's implement it. We have the following changes in the code:

```
static void PerformSingleSwap(List<Card> cards, int index)
{
    int randomIndex = rand.Next(1, cards.Count);
    Card firstCard = cards[index];
    Card randomCard = cards[randomIndex];
    cards[index] = randomCard;
    cards[randomIndex] = firstCard;
}
```

```
static void ShuffleCards(List<Card> cards)
{
  for (int i = 0; i < cards.Count; i++)
  {
    PerformSingleSwap(cards, i);
  }
}
```

We execute the program and get **a better shuffle of the deck** of 52 cards:

```
(9 HEART)(5 CLUB)(3 CLUB)(7 SPADE)(6 CLUB)(5 SPADE)(6 HEART)(4
CLUB)(10 CLUB)(3 SPADE)(K DIAMOND)(10 HEART)(8 CLUB)(A CLUB)(J
DIAMOND)(K SPADE)(9 SPADE)(7 CLUB)(10 DIAMOND)(9 DIAMOND)(8
HEART)(6 DIAMOND)(8 SPADE)(5 DIAMOND)(4 HEART)(10 SPADE)(J
CLUB)(Q SPADE)(9 CLUB)(J HEART)(K CLUB)(2 HEART)(7 HEART)(A
HEART)(3 DIAMOND)(K HEART)(A SPADE)(8 DIAMOND)(4 SPADE)(3
HEART)(5 HEART)(Q HEART)(4 DIAMOND)(2 SPADE)(A DIAMOND)(2
DIAMOND)(J SPADE)(7 DIAMOND)(Q DIAMOND)(2 CLUB)(6 SPADE)(Q
CLUB)
```

It looks like **the cards are finally randomly ordered** at every execution of our program. For now we don't see any bugs (i.e. repeating or missing cards, or cards, which are at the same place). The program is fast and it doesn't fall asleep. It looks like we have done well.

Let's take a look at the other sample task: **sorting numbers**. What would be a serious common-case test? Easy for us would be to generate 100 or even **1000 random numbers** and sort them. It is easy to check whether the final answer is correct: the numbers must be sorted by size. Another good test would be to enter **the numbers from 1000 to 1 in descending order**. The output must also consist of 1000 numbers, but sorted in ascending order. We could say that the hardest test is to check whether our program can handle many numbers – then we could say that our program probably works correctly.

Let's take a look at some other types of tests, which we must consider when we solve programming problems.

Borderline Cases

The step, we often miss in problem solving, is testing for **borderline cases**. Borderline situations occur when the input data is on the border between a normal situation and a situation, which most likely will not happen. In this situation the program often crashes, because very large or very small amounts of data are not considered, although they are possible to be entered. This is clearly a programmer's fault, because he has not thought that this could happen.

How to think up borderline cases? We analyze all of the data, which is being entered, to our program and think up such **extreme values**, which are possible to be entered. These values could be **extremely small**, **extremely large** or **just strange**. If it is said that the upper limit is 52 cards, the values around 52 are also borderlines and they could cause errors.

Borderline Case: Shuffling One Card

In our shuffling-cards problem a borderline case is to **shuffle only one card**. This case is absolutely valid (although it is quite unusual), but our program may not handle it correctly. Let's take a look what could happen if we enter a deck of one card. We could write the following little test:

```
static void TestShuffleOneCard()
{
  List<Card> cards = new List<Card>();
  cards.Add(new Card() { Face = "A", Suit = Suit.CLUB });
  CardsShuffle.ShuffleCards(cards);
  CardsShuffle.PrintCards(cards);
}
```

We execute our program and get an **unexpected result**:

```
Unhandled Exception: System.ArgumentOutOfRangeException: Index
was out of range. Must be non-negative and less than the size of
the collection. Parameter name: index
  at
System.ThrowHelper.ThrowArgumentOutOfRangeException(ExceptionArg
ument argument, ExceptionResource resource)
  at System.ThrowHelper.ThrowArgumentOutOfRangeException()
  at System.Collections.Generic.List`1.get_Item(Int32 index)
  at CardsShuffle.PerformSingleSwap(List`1 cards, Int32 index)
in D:\Projects\Cards\CardsShuffle.cs:line 61
…
```

The error occurred because the arguments passed to the method, which generates random numbers, were invalid. Our program can handle a normal deck of cards, but it **can't handle a deck of one card**. We found an easy fix for this bug, which we could miss lightly, if we skipped checking the borderline cases. After we have established the nature of the problem, we can fix it:

```
static void ShuffleCards(List<Card> cards)
{
  if (cards.Count > 1)
  {
    for (int i = 0; i < cards.Count; i++)
    {
```

```
        PerformSingleSwap(cards, i);
      }
    }
}
```

We test again and we are sure that **the bug is fixed**.

Borderline Case: Shuffling Two Cards

After there is a problem with one card, there may be also a problem **with two cards**. It sounds logical, does it? It doesn't bother us to test it. We run the program several times with only two cards to be shuffled and expect to get a different order on every run. Here is an example source code that will do the trick:

```
static void TestShuffleTwoCards()
{
  List<Card> cards = new List<Card>();
  cards.Add(new Card() { Face = "A", Suit = Suit.CLUB });
  cards.Add(new Card() { Face = "3", Suit = Suit.DIAMOND });
  CardsShuffle.ShuffleCards(cards);
  CardsShuffle.PrintCards(cards);
}
```

We run the program several times and **the output is always the same**:

```
(3 DIAMOND)(A CLUB)
```

It seems **something is still not right**. If we take a look at the source code or run the debugger we notice that two swaps are made: this first card with the second and immediately after that the second with the first one. The result is one and the same. How to solve this problem? Instantly we can think up a few solutions:

- We perform N+K single swaps, where K is a random number between 0 and 1.
- We assume that the random position can be also a zero position.
- We consider a deck of 2 cards as a special case and write a separate method to handle this case.

This second solution is the simplest to be implemented. Let's try it. Here is the source code:

```
static void PerformSingleSwap(List<Card> cards, int index)
{
  int randomIndex = rand.Next(0, cards.Count);
  Card firstCard = cards[index];
```

```
    Card randomCard = cards[randomIndex];
    cards[index] = randomCard;
    cards[randomIndex] = firstCard;
}
```

We test again and it **looks like the program works correctly**: sometimes the cards are shuffled and sometimes not.

If there is a problem with 2 cards, there may be also a problem with 3 cards, right? If we do a test with 3 cards we see that it works flawlessly. After a few runs we get every possible order of the cards (all permutations of the cards). This time we did not find any bugs and we do not need to edit the code.

Borderline Case: Shuffling Zero Cards

What other tests do we need to do? Are there other **unusual, borderline cases**? Let's think. What is going to happen if we sh**uffle an empty list of cards**? This really is a bit strange, but a program must work correctly or it must properly alert for an error. Let's look at result of our program. The result is an empty list. Is this correct? Yes, if we try to shuffle zero cards the answer should be again a deck of zero cards. Everything looks fine.

With the input of invalid data the program must not return an incorrect result. It must return a correct result or alert that the input data is wrong.

What do you think about the rule above? It is logical, right? Imagine a program, which displays images. What is going happen if we input an image, which is actually an empty file? This is an unusual situation, which should not happen, but it may happen. If with the input of an empty file the program crashes or throws an unprocessed exception, this would be annoying for the user. Normally, instead of an empty file, a special image should be displayed or an alert should pop-up, which contains that the image is invalid.

Think about **how many borderline cases are there in Windows**. What happens if we try to print an empty file? Does Windows crash or the "the blue screen of death" appears? What happens if in the Windows' calculator we try divide by zero? What happens if we try to copy an empty file (with size of 0 bytes) using Windows Explorer? What happens if try to save a file without a name in Notepad (with an empty string as a name)? You see that there are many borderline cases. Our duty as programmers is to **fix the bugs before the product is released on the market**.

Let's go back to the card-shuffling problem. Thinking about other borderline and unusual cases, we consider a case with -1 cards to be shuffled? Because we cannot initialize an array of -1 elements, we consider it as incorrect.

We don't have an upper bound, so we don't have any other special cases (similar to the case with 1 card). We stop searching for borderline cases

having to do with the count of the cards. It looks like we considered all of the cases.

Now we only need to check whether there are such **values of the input, which can cause errors** – an invalid suit, a card with negative value (i.e. -1 club). Actually our algorithm does not care about what is being shuffled (cards or books), so this should not be a problem. If we have any doubts, we could do a test and be sure, that even with an invalid deck of cards, the answer is correct.

We look around for other borderline case in the input data, but we could not think up any. We only need to do a performance test, right? Actually we forgot to test our program again after the corrections.

Regression Testing

While fixing bugs, we often create new bugs without to notice. For example if we fix a bug for two cards with editing the method, responsible for the shuffling of the cards, a new bug with 3 cards may occur. On every edit of code, which concerns other cases, we must again do the same test, we have done earlier. That's why it is a good idea to **save the tests as methods** (naming them with a prefix **`Test`**) and be able to run them again. Re-testing with the tests already passed in the past is called "**regression testing**". We may also use the unit testing framework that comes with Visual Studio (see Unit Testing section in the "High-Quality Code" chapter) to simplify re-running the tests after making changes in the code.

The idea of **repeating the tests** is implicit in the concept of unit testing. As we mentioned earlier, this concept is for more advanced programmers.

After the changes we need to try again shuffling 0 cards, 1 card, 2 cards, 3 cards and 52 cards.

When you have found and fixed in your code a bug, which concerns a specific test, make sure that no other bugs have been introduced after the code has been changed. It is a good idea to keep the tests saved for repeated execution.

Performance Tests

It is normal to have some performance requirements about a module or the program at all. No one likes his computer to be slow, right? That's why you have to write software, which works slow only if there is a good reason.

How to check the speed (performance) of our program? We must firstly consider whether we have any performance requirements. Then what are they? If we don't have any, we should use some conventional criteria.

Shuffling Cards – Performance Tests

Let's take a look at our deck-shuffling program. What are the **performance criteria**? Do we have such? We don't have any requirements like: "the program must compute the answer in less than a second with an upper bound of 500 cards on a modern computer". After we don't have such explicit criteria, we should **set our own**.

Because the data is a set of cards, we consider testing with a normal deck of 52 cards. We already ran such a test and the answer was output immediately. It looks like our program works fast in normal conditions.

It is normal to test also with a deck of many more cards (i.e. 52,000). In a very particular situation someone might like to **shuffle a deck of 52,000 cards** and we could not allow our program to crash at this later phase. We can easily create such a test by adding 1000 times our 52 cards. Here is some example code:

```
static void TestShuffle52000Cards()
{
   List<Card> cards = new List<Card>();
   string[] allFaces = new string[] {"2", "3", "4", "5",
      "6", "7", "8", "9", "10", "J", "Q", "K", "A"};
   Suit[] allSuits = new Suit[] { Suit.CLUB, Suit.DIAMOND,
      Suit.HEART, Suit.SPADE};
   for (int i = 0; i < 1000; i++)
   {
      foreach (string face in allFaces)
      {
         foreach (Suit suit in allSuits)
         {
            Card card = new Card() { Face = face, Suit = suit };
            cards.Add(card);
         }
      }
   }
   ShuffleCards(cards);
   PrintCards(cards);
}
```

We start the program and notice that it **works without stopping for 5-10 seconds**. Of course with slower computers it is going to take more time. What is happening? The count of the operations with a deck of 52,000 cards is almost the same as with a deck of 52 cards for example. Why is the program so slow? Advanced programmers will immediately guess that the delay is because **we output big amounts of data** and **printing to the console is a slow operation**. If we comment the line, where the printing is done, as a

comment, we will see how fast our program is, even with a deck of 52,000 cards. Here's how we can count the time:

```
static void TestShuffle52000Cards()
{
  …
  DateTime oldTime = DateTime.Now;
  ShuffleCards(cards);
  DateTime newTime = DateTime.Now;
  Console.WriteLine("Execution time: {0}", newTime - oldTime);
  //PrintCards(cards);
}
```

We can check how long it takes for the method, responsible for the shuffling of the cards, to be executed:

```
Execution time: 00:00:00.0156250
```

One millisecond is **absolutely acceptable**. We don't have any performance problems.

Sorting Numbers – Performance Tests

Let's take a look at another one of our problems: **sorting an array of numbers**. Here performance might be a problem. We have thought up a simple solution: our algorithm finds the lowest number in the array and swaps it with the number at position 0. Then it finds the next lowest number and sets it at position 1. The algorithm continues until we reach the last position. We will not comment the correctness of the algorithm. It is well known as "**Selection sort**" (http://en.wikipedia.org/wiki/Selection_sort).

Let's suppose that we have passed all the steps for solving a programming problem, except the last one. Therefore we will try to **sort 10,000 random numbers**:

Sort10000Numbers.cs

```
using System;

public class Sort10000Numbers
{
  static void Main()
  {
    int[] numbers = new int[10000];
    Random rnd = new Random();
    for (int i = 0; i < numbers.Length; i++)
    {
```

```
            numbers[i] = rnd.Next(2 * numbers.Length);
        }
        SortNumbers(numbers);
        PrintNumbers(numbers);
    }

    static void SortNumbers(int[] numbers)
    {
        for (int i = 0; i < numbers.Length - 1; i++)
        {
            int minIndex = i;
            for (int j = i + 1; j < numbers.Length; j++)
            {
                if (numbers[j] < numbers[minIndex])
                {
                    minIndex = j;
                }
            }

            int oldNumber = numbers[i];
            numbers[i] = numbers[minIndex];
            numbers[minIndex] = oldNumber;
        }
    }

    static void PrintNumbers(int[] numbers)
    {
        Console.Write("[");
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.Write(numbers[i]);
            if (i < numbers.Length - 1)
            {
                Console.Write(", ");
            }
        }
        Console.WriteLine("]");
    }
}
```

We run our program and it looks like it **finishes in less than a second** on a typical modern computer. The shortened result would be something like that:

```
[0, 14, 19, 20, 20, 22, …, 19990, 19993, 19995, 19996]
```

We do another test, this time with an array of **300,000 random numbers**, and notice that our program **falls asleep** or it is just too slow. This is a **serious performance problem**.

Before we start fixing it, we must ask ourselves: is this a real situation. If we sort students' marks, the elements will not be more than a couple of dozens. However if we sort the stocks' currencies of a large software company for its being on the stock market, we have a lot of numbers to be sorted. This is because the price of the stocks can change every second. In about ten years the price can change hundreds of thousands times of times. In this case we should **look for a better sorting algorithm**.

We can easily find information about sorting algorithms in many websites and books. In our case, it is most appropriate to use the non-comparative integer sorting algorithm "**radix sort**" (en.wikipedia.org/wiki/Radix_sort) which runs in linear time, but this discussion is beyond the objectives of this book.

Let's recall the efficiency rule:

You must always make a compromise between the time, spent on writing the program, and the performance, which we want to achieve. Otherwise you might lose your time for solving a problem, that doesn't exist, or come up with a solution, which is inefficient.

We must consider that for some problems there aren't any fast algorithms. For example **there aren't any fast solutions** for finding all prime dividers of an integer (take a look at http://en.wikipedia.org/wiki/Integer_factorization).

In some situations the input data is too small and we don't need a fast algorithm to process it. For example: sorting the students' marks. It can be solved with every sorting algorithm, because the count of the students is small.

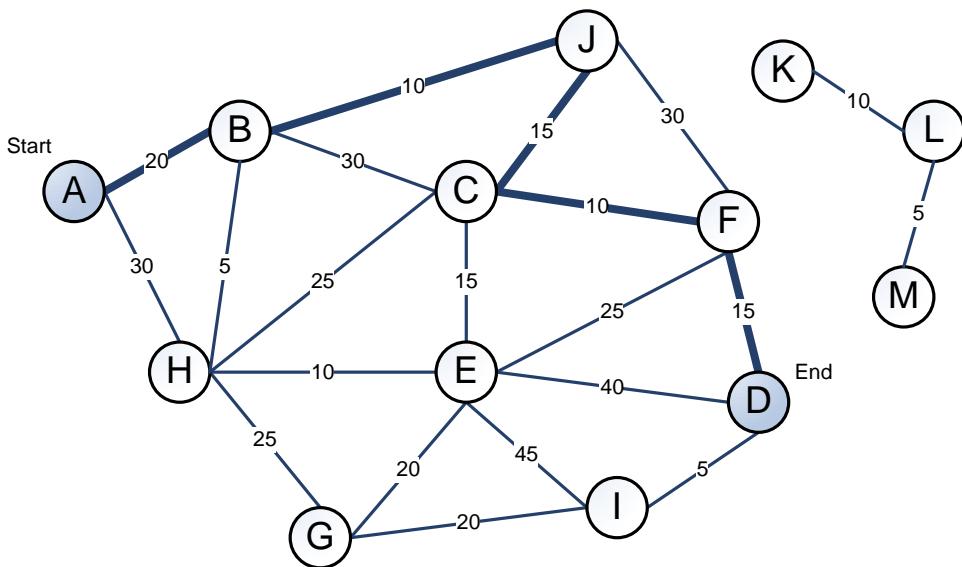
General Conclusions

Before you have started reading this chapter, you have probably thought that it is going to be boring. We believe that you think of this chapter in a different way now. Many people think that they can solve programming problems and that there is no recommended approach (you just have to do it). Unfortunately, there are lots of approaches for solving problems. We did not only tell you how, **we showed you how**! We convinced you that our approach can give good results, right?

Just **think how many bugs we found** while we were solving quite an easy problem: shuffling a deck of cards. Would we have written an efficient solution, if we had not considered the steps above? And what will happen if we try to solve a more difficult problem, i.e. finding the optimal path through the traffic jams in New York with a map and current statistics given? It's absolutely insane to try the first idea you come up with without considering many more. The first step in acquiring programming-problem-solving

techniques is to **learn to approach the problem systematically** and to acquire the recipe for problem solving, we demonstrated earlier. Of course, this is not enough, but it is a crucial step!

There is a recipe for programming-problem solving! Use a systematical approach for better results, instead of your sense. Even professionals, with more than 10 years of experience, use the approach described in this chapter. Use it yourself and you will be convinced that it works! Don't forget to test your solution seriously and deeply.

Finally we want to take a note on the cards shuffle algorithm. The "**cards shuffling**" is well-known problem in computer science and there are classic algorithms for solving it like the "**Fisher-Yates Shuffle**". Read more in Wikipedia: http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle.

Exercises

1. Using the described in this chapter **methodology of solving programming problems**, solve this problem: In a plane **N points** are given (**N** < 100,000). The coordinates of the points are integers (x_i, y_i). Write a program, which **finds all horizontal and vertical lines**, such that they split the plane into two parts, each containing an equal set of points (points lying on the line are note counted).

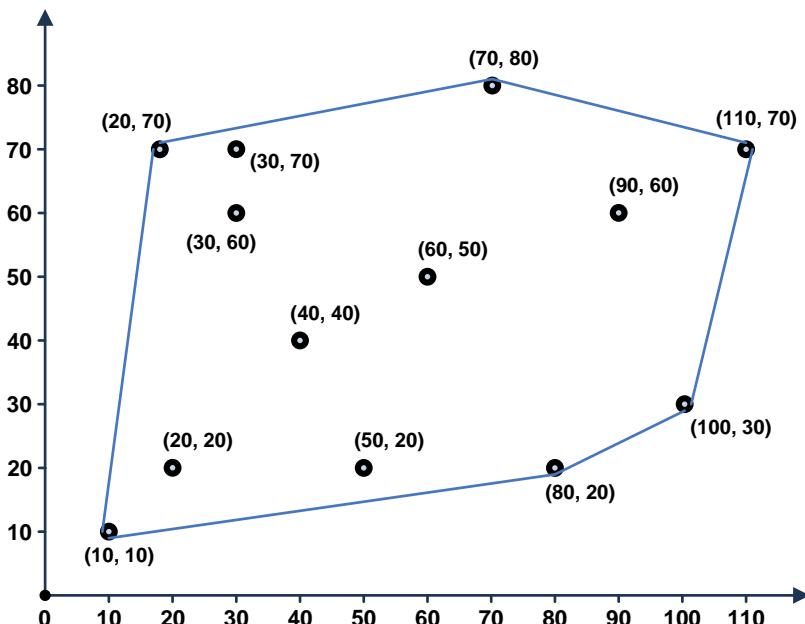
2. Using the described in this chapter methodology of solving programming problems, solve this problem: A set **S** of **n** integers and a positive integer **k (k ≤ n ≤ 10)** are given. An **alternating sequence** is a sequence, which changes its behavior from ascending to descending and vice versa after every element. Write a program, which generates all possible sequences $s_1, s_2, \ldots, s_k$ containing **k** different elements from **S**.

   Example: **S = { 2, 5, 3, 4 }**, **K = 3**: {2, 4, 3}, {2, 5, 3}, {2, 5, 4}, {3, 2, 4}, {3, 2, 5}, {3, 4, 2}, {3, 5, 2}, {3, 5, 4}, {4, 2, 3}, {4, 2, 5}, {4, 3, 5}, {4, 5, 2}, {4, 5, 3}, {5, 2, 3}, {5, 2, 4}, {5, 3, 4}.

3. Using the described **methodology** of creating solutions to programming problems, solve the following problem: a **map of a city** is given. At this map there are given **roads and crossroads**. For every road a **length** is given. One crossroad can connect a couple of roads. Your program must **find the shortest path from one crossroad to another** (the shortest path is measured as the sum of the lengths of all includes roads).

   A **sample map** is given below. At this map the shortest path between the crossings **A** and **D** has **length of 70** and it is shown on the figure with bold lines. As you can see, this is not the only path from **A** to **D**: there are more paths with different lengths. Note that not always the first shortest road considered as current next node leads to finding the shortest path, neither does the lowest count of roads. Between some crossings there may not even be a road. This creates a very interesting problem.

The **input data** is contained in the file **map.txt**. A list of the roads and their length are given first, followed by a blank line and the pairs of crossroads, between which we need to find the shortest path. At the end of the file a blank line is given. Crossroads are denoted by a single letter of sequence of letters. Here is a sample input file:

```
A B 20
A H 30
B H 5
…
L M 5
(blank line)
A D
H K
A E
(blank line)
```

The **result** from the execution of your program should be printed on the console in the following format: for every pair of crossroads from the input file, the **shortest path length** should be printed, followed by the **path itself**. For the map above the output will look like this:

```
70 ABJCFD
No path!
35 ABHE
```

4. * In a plane N points are given with integer positive coordinates. These points represent **trees in a field**. The gardener wants to **enclose all trees using a minimum amount of fence**. Write a program, which

finds the appropriate points, through which the fence goes. Use the methodology of solving programming problems!

For example the garden may look like this:

The **input data** is read from the text file **garden.txt**. On the first line we are given **N** – the number of points is given. Next we are given N pairs – the coordinates of the points. For our example garden the file contains the following data:

```
13
60 50
100 30
40 40
20 70
50 20
30 70
10 10
110 70
90 60
80 20
70 80
20 20
30 60
```

The **output data** must be printed to the console as a sequence of points, through which the fence goes. Here is an output example:

```
(10, 10) - (20, 70) - (70, 80) - (110, 70) - (100, 30) -
(80, 20) - (10, 10)
```

Solutions and Guidelines

1. First find a **sheet of paper**, on which you can draw the coordinate system and the points. In this way you can easily think up a solution and consider some **tests**. Consider different solutions and compare them in terms of quality, performance and time needed to be implemented. Hint: you may **sort** the points by **X** or **Y** and find the lines with a **linear scan**.

2. Again take a pencil and a sheet. Write down many examples and consider them. What ideas do you come up with? Do you need more examples? Think over your ideas, write them down on a sheet of paper, if you are sure they are correct – implement them. Think for examples that your program may not handle. It is always a good idea to think up specific examples and then to implement them. Think how your solution will work with different values of **K** and different values and count of elements in **S**.

3. Use the described **methodology for creating solutions** to programming problems! The problem is difficult and you will have to spend more time on it. First **draw the map on a sheet of paper**. Try to think up a correct algorithm for finding the shortest path yourself. Then search for "shortest path algorithm" in the Internet. Most likely you will be able to find some article about the problem quickly.

   Check whether the algorithm is correct. **Test** with different cases.

   In what **data structure** are you going to keep the map of the city? Think about which are the operations your algorithm is going to use. Probably you will come up with idea to keep a **list of the roads for every crossroad** and the crossroads in a **hash table**.

   Think about the **performance**. Will your algorithm handle 1,000 crossings and 5,000 roads fast enough?

   Write the code **step by step**. First write the method, which reads the input file. In this case the data is entered from a file and therefore you can consider it as your first step. If the data was entered from the console, we would have to leave it as our last step. Then implement the algorithm for finding the minimum path length. If you can, **break up the implementation into smaller steps**. For example you could firstly find the length of the minimal path without the path itself, because it is easier. Then implement the finding of the shortest path itself (as a list of crossroads). Think what could happen if there are **several shortest paths**. Finally implement the output of the result.

   Test your solution! Test with an empty map. Test with a map with one crossroad. Test with an example, in which there are no roads between the crossroads. Test with a big map (1,000 crossings and 5,000 roads). You

can generate such a case with a small program. For the names of the crossroads you have to use **string** instead of **char**, right? If you use char, it will not be possible to have 1,000 crossings? Is your solution fast? Does it work correctly?

Be careful with the input and output data formats. Comply with the described format requirements in the problem statement!

4. If you are not very good with **analytic geometry**, you may not come up with a correct solution by yourself. Search in the Internet for "**convex hull algorithm**". Knowing that the fence is called a "convex hull" of a set of points in the plane, you will be able to find hundreds of articles – even some with source code in C#. Don't copy others' mistakes, especially the source code! **Think!** Study how the algorithm works.

   Check whether the algorithm is correct. **Test with different examples** (firstly on a **sheet of paper**). What could happen if there are a couple of points, lying on a line in the plane? Do you need to include all of them? Think what could happen if there is a couple of convex hulls. From which point will the hull start? Will you move clockwise or counterclockwise? Is there a requirement in the description of the task how must the points be ordered? What if some of the points are the same? What should happen if we have no points or just one point in the field?

   In what **data structure** will you keep the points? In what data structure will you keep the **convex hull**?

   Think about the **performance**. Will your algorithm work for 1,000 points?

   Write your program **step by step**. First implement the reading of the input data. Implement your convex hull finding algorithm. If it is possible, **break your algorithm into smaller parts**. Finally implement the output of the result in the described format.

   Test your solution! What will happen if we have 0 points? Try with 1 point. Try with 2 points. Try with 5 point, which lie on a line. Does your algorithm work? What will happen if we have 10 points and another 10, which match the first 10? What will happen if we have 10 points, which lie one over another? What will happen if you have many points, i.e. 1,000? Does your algorithm work fast? What will happen if the coordinates of the points are large numbers (100,000,000; 200,000,000)? Does this affect your algorithm? Can you face precision errors in the calculations?

   Be careful with the input and output data. Consider the format in the проблем description! Do not invent your own format of the input and output data. The formats are defined and they must be respected.

   If you want, you could implement **visualization of the points** and the convex hull using **Windows Forms** or in **WPF**. You may implement a **generator of random tests** and test your solution many times by observing the visualization of the hull – does it really wrap the points.

Chapter 24. Sample Programming Exam – Topic #1

In This Chapter

In this chapter we will look at and offer **solutions to three problems from a sample programming exam**. While solving them we will put into practice the methodology described in the chapter "Methodology of Problem Solving".

Problem 1: Extract Text from HTML Document

We are given HTML file named **`Problem1.html`**. Write a program, which **removes all HTML tags** and retains only the text inside them. Output should be written into the file **`Problem1.txt`**.

Sample input file for **`Problem1.html`**:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Sample output file for **`Problem1.txt`**:

```
Welcome to our site!
Home
Contacts
About
```

Inventing an Idea

The first thing that occurs to us as an **idea for solving this problem** is to read sequentially (e.g. line by line) the input file and to remove any tags. It is easily seen that all tags starting with the character "**<**" and end with the character "**>**". This also applies to opening and closing tags. This means that for each line in the file we should remove all substrings starting with "**<**" and ending with "**>**".

Checking the Idea

We have an idea for solving the problem. Whether the idea is correct? First we **should check it**. We can ensure it is correct for the sample input file, and then consider whether there are specific cases where the idea could be incorrect.

We **take a pen and paper** and check by manually whether the idea is correct. We do this by striking out all text substrings that start with the character "**<**" and end with the character "**>**". As we do so, we see that there is only pure text and any tags disappear:

```
<html>
<head><title>Welcome to our site!</title></head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a>
<a href="/contacts.html">Contacts</a>
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Now we have to think of some **more special cases**. We do not want to write 200 lines of code and only then think about special case, finding out we have to redesign the entire program. It is important to check the problematic situations now, before we begin writing the code of the solution. We can think of the following special example:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

There are two things to consider:

- There are tags containing text that **open and close at separate lines**. Such tags in our example are **<html>**, **<body>** and **<a>**.
- There are tags that contain text and other tags in themselves (**nested tags**). For example **<body>** and **<html>**.

What should be the result of this example? If we directly remove all tags we will get something like this:

```
Clickon this
linkfor more info.
This isboldtext.
```

Or maybe we should follow the rules of the HTML language and get the following result:

```
Click on this link for more info.
This is bold text.
```

There are other options, such as putting each piece of text, which is not a tag, on a new line:

```
Click
on this
link
for more info.
This is
bold
text.
```

If we remove all the text in tags and snap the other text, we will get **words that are stuck together**. From the task's description it is not clear if this is the requested result or it must be as in the HTML language. In the HTML language each series of separators (spaces, new lines, tabs, etc.) appear as a space. However, this was not mentioned in the task's description it is not clear from the sample input and output.

It is not clear yet whether to print the words that are in a tag which holds other tags or to skip them. If we print only the contents of the tags, which consist of text only, we will get something like this:

```
on this
link
bold
```

It is yet not clear from the description, how to display the text that is located on a few lines inside a tag.

Clarification of the Statement of the Problem

The first thing to do when we find ambiguity in the description of the task is to **read it carefully**. In this case the problem statement is not really clear and does not give us the answers. Probably we should not follow the HTML rules, because they are not described in the problem statement, but it is not clear whether to connect the words in neighboring tags or separate them by a space or new line.

This leaves us only one thing – **to ask**. If we have an exam, we will ask the one who gives us the task. In real life, someone is an **owner** of the software we develop, and he could **answer the questions**. If nobody can give an answer, choose one option that seems most correct under the information we have and act on it. Assume that we need to print text, which remains after removing all opening and closing tags, **using a blank line separator at the positions of the tags**. If there are blank lines in the text, we keep them. For our example, we should obtain the following **correct output**:

```
Click
on this
link
for more info.
This is
bold
text.
```

A New Idea for Solving the Problem

So, after adapting these **new requirements**, the following idea comes: read file line by line and **substitute each tag with a new line**. To avoid duplication of new lines in the resulting file, replace every two consecutive lines of new results with a new line.

We **check the new idea** with the example from the original statement of the problem with our example to ensure it is correct. It remains to implement it.

Break a Task into Subtasks

The task can easily break into **3 subtasks**:

- **Read** the input file.
- **Processing** of a line of input file: replace tags with a new line.
- **Print** results in the output file.

What Data Structures to Use?

In this task we must perform simple word processing and file management. The question of what data structures to use is not a problem – **for word processing** we use **string** and if necessary – **StringBuilder**.

Consider the Efficiency

If we read the lines one by one, it will not be a slow operation. Processing of one line can be done by replacing some characters with others – a quick operation. We should not have performance problems.

A possible problem could be the clearing of the empty lines. If we collect all lines in a buffer **(StringBuilder)** and then remove double blank lines, this buffer will occupy too much memory for large input files (for example 500 MB input file).

To save memory, we will try to clean the excess blank lines just after the replacement tags with the white space character.

Now we examined the idea of solving the task, we ensured that it is good and covers the special cases that may arise, and believe **we will have no performance issues**.

Now we can safely **proceed to implementation** of the algorithm. We will write the code step by step to find errors as early as possible.

Step 1 – Read the Input File

The first step solving the given task is **reading the input file**. In our case it is a HTML file. This should not bother us, because HTML is a text format. Therefore, to read it, we will use the **StreamReader** class. We will traverse the input file line by line and each line we will derive (for now it is not important how we will do it) all the information we need (if any) and save it into an object of type **StringBuilder**. Extraction we will implement in the next step (step 2). Let's write the necessary code for the implementation of our first step:

```
string line = string.Empty;
StreamReader reader = new StreamReader("Problem1.html");

while ((line = reader.ReadLine()) != null)
{
  // Find what we need and save it in the result
}

reader.Close();
```

With this code we will read the input file line by line. Let's think whether we have completed a good first step. **Do you know what we have missed?**

From the code written we will read the input file, but only if it exists. **What if the input file does not exist** or could not be opened for some reason? Our present decision does not deal with these problems. There is another problem in our code too: if an error occurs while reading or processing the data file, it **will not be closed**.

With **File.Exists(…)** we will check if the input file exists. If not – we will display an appropriate message and stop program execution. To avoid the second problem we will use the **try-catch-finally** statement (we may use the **using** statement in C# as well). Thus, if an exception arises, we will process it and will always close the file, which we worked with. We must not forget that the object of the **StreamReader** class must be declared outside the **try** block, otherwise it will be unavailable in the **finally** block. This is not a fatal error, but often made by novice programmers.

It is better to define the **input file name as a constant**, because we probably will use it in several places.

Another thing: when reading from a text file it is appropriate to specify explicitly the **character encoding**. Let's see what we get:

```
using System;
using System.IO;
using System.Text;

class HtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string Charset = "windows-1251";

   static void Main()
   {
      if (!File.Exists(InputFileName))
      {
         Console.WriteLine(
            "File " + InputFileName + " not found.");
         return;
      }

      StreamReader reader = null;
      try
      {
         Encoding encoding = Encoding.GetEncoding(Charset);
         reader = new StreamReader(InputFileName, encoding);
         string line;
         while ((line = reader.ReadLine()) != null)
         {
```

```
            // Find what we need and save it in the result
          }
        }
        catch (IOException)
        {
          Console.WriteLine(
            "Can not read file " + InputFileName + ".");
        }
        finally
        {
          if (reader != null)
          {
            reader.Close();
          }
        }
    }
}
```

Test the Input File Reading Code

We handled the described problems and it **seems we have implemented the reading of the input file**. We wrote a lot of code. To be convinced that it is correct, we can **test our unfinished code**. For example let's print the content of the input file to the console, and then try processing nonexistent files. The writing will be done in a **while** loop using **Console.WriteLine(…)**:

```
…
while ((line = reader.ReadLine()) != null)
{
  Console.WriteLine(line);
}
…
```

If we test the piece of code we have with the **Problem1.html** sample file from the problem description, the result is correct – the input file itself:

```
<html>
<head>
<title>Welcome to our site!</title>
</head>
<body>
<center>
<img src="/en/img/logo.gif" width="130" height="70" alt="Logo">
<br><br><br>
<font size="-1"><a href="/index.html">Home</a> -
```

```
<a href="/contenst.html">Contacts</a> -
<a href="/about.html">About</a></font><p>
</center>
</body>
</html>
```

Let's try a **nonexistent file**. We change the file name **Problem1.html** with **Problem2.html**. The result is the following:

```
File Problem2.html not found
```

We are convinced that **the code till now is correct**. Let's move to the next step of the implementation of our idea (algorithm).

Step 2 – Remove the Tags

Now we want to find a suitable way to **remove all tags**. How should we do this?

One possible way is to **check the line character by character**. For each character in the current row we will look for the character "**<**". On the right side of it we will know that we have a tag (opening or closing). The end tag character is "**>**". So we can find tags and remove them. To not get the words connected between adjacent tags, each tag will be replaced with the character for a blank line "**\n**".

The algorithm is simple to implement, but isn't there a more clever way? Can we use **regular expressions**? They can easily look for tags and replace them with "**\n**", right? In the same time the code will be simple and in case of errors, they will be removed more easily. We will consider this option. What should we do? First we need to write a regular expression. Here is how it may look:

```
<[^>]*>
```

The idea is simple: any string, that starts with "**<**", continues with arbitrary sequence of characters, other than "**>**" and ends with "**>**" is an HTML tag. Here's how we can **replace the tags with a new line**:

```
private static string RemoveAllTags(string str)
{
  string textWithoutTags = Regex.Replace(str, "<[^>]*>", "\n");
  return textWithoutTags;
}
```

After coding this step, we should test it. For this purpose again we print to the console the strings we found via **Console.WriteLine(…)**. And test the code:

HtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

class HtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string Charset = "windows-1251";

   static void Main()
   {
      if (!File.Exists(InputFileName))
      {
         Console.WriteLine(
            "File " + InputFileName + " not found.");
         return;
      }

      StreamReader reader = null;
      try
      {
         Encoding encoding = Encoding.GetEncoding(Charset);
         reader = new StreamReader(InputFileName, encoding);
         string line;
         while ((line = reader.ReadLine()) != null)
         {
            line = RemoveAllTags(line);
            Console.WriteLine(line);
         }
      }
      catch (IOException)
      {
         Console.WriteLine(
            "Can not read file " + InputFileName + ".");
      }
      finally
      {
         if (reader != null)
         {
            reader.Close();
         }
```

```
        }
    }

    private static string RemoveAllTags(string str)
    {
        string strWithoutTags =
            Regex.Replace(str, "<[^>]*>", "\n");
        return strWithoutTags;
    }
}
```

Testing the Tag Removal Code

Let's **test the program** with the following input file:

```
<html><body>
Click<a href="info.html">on this
link</a>for more info.<br />
This is<b>bold</b>text.
</body></html>
```

The result is as follows:

```
(empty rows)
Click
on this
link
for more info.
(empty row)
This is
bold
text.
(empty rows)
```

Everything **works perfectly**, only that we have **extra blank lines**. Can we remove them? This will be our next step.

Step 3 – Remove the Empty Lines

We can remove unnecessary blank lines, replacing a double blank line "**\n\n**" with a single blank line "**\n**". We should not have groups of more than one character for a new line "**\n**". Here is an example how we can perform the substitution:

```
private static string RemoveDoubleNewLines(string str)
```

```
{
  return str.Replace("\n\n", "\n");
}
```

Testing the Empty Lines Removal Code

As always, before we move forward, we test whether the method works correctly. We try a text, which has no blank rows, and then add 2, 3, 4 and 5 blank lines, including at the beginning and at the end of text.

We find that **the above method does not work correctly**, when there are 4 blank lines one after another. For example, if we submit as input "**ab\n\n\n\ncd**", we will get "**ab\n\n\cd**" instead of "**ab\ncd**". This defect occurs because the **Replace(…)** finds and replaces a single match, scanning the text from left to right. If in result of a substitution the searched string reappears, it is skipped.

See how useful it is when each method is tested on time. We do not end up wondering why the program does not work when we have 200 lines of code, full of errors. **Early detection of defects is very useful** and we should do it whenever possible. Here is the corrected code:

```
private static string RemoveDoubleNewLines(string str)
{
  string pattern = "[\n]+";
  return Regex.Replace(str, pattern, "\n");
}
```

The above code uses a regular expression to find any sequence of **\n** characters and replaces it with a single **\n**.

After a series of tests, we are **convinced that the method works correctly**. We are ready to test the program that removes all unnecessary newlines. For this purpose we make the following changes:

```
while ((line = reader.ReadLine()) != null)
{
  line = RemoveAllTags(line);
  line = RemoveDoubleNewLines(line);
  Console.WriteLine(line);
}
```

We test the code again. Still it **seems there are blank lines**. Where do they come from? Perhaps, if we have a line that contains only tags, it will cause a problem. Therefore we may prevent this case. We add the following checks:

```
if (!string.IsNullOrEmpty(line))
```

```
{
  Console.WriteLine(line);
}
```

This removes most of the blank lines, but not all.

Remove the Empty Lines: Second Attempt

If we think more, it could happen so, that **a line begins or ends with a tag**. Then this tag will be replaced with a single blank line and so at the beginning or at the end of the line we may get a blank line. This means that we should clean the empty rows at the beginning and at the end of each line. Here's how we can make the cleaning:

```
private static string TrimNewLines(string str)
{
  int start = 0;
  while (start < str.Length && str[start] == '\n')
  {
    start++;
  }

  int end = str.Length - 1;
  while (end >= 0 && str[end] == '\n')
  {
    end--;
  }

  if (start > end)
  {
    return string.Empty;
  }

  string trimmed = str.Substring(start, end - start + 1);
  return trimmed;
}
```

The method works very simply: goes from left to right and **skips all newline characters**. Then passes from right to left and skips again all newline characters. If the left and right positions have passed each other, this means that the string is either empty or contains only newlines. Then the method returns an empty string. Otherwise it returns back everything to the right of the start position and to the left of the end position.

Remove the Empty Lines: Test Again

As always, we **test whether the above method works correctly** with several examples, including an empty string, no string breaks, string breaks left or right or both sides and a string with new lines. We make sure, that **the method now works correctly**.

Now we have to modify the logic of processing the input file:

```
while ((line = reader.ReadLine()) != null)
{
    line = RemoveAllTags(line);
    line = RemoveDoubleNewLines(line);
    line = TrimNewLines(line);
    if (!string.IsNullOrEmpty(line))
    {
        Console.WriteLine(line);
    }
}
```

Step 4 – Print Results in a File

It remains to **print the results in the output file**. To print the results in the output file we will use the **`StreamWriter`**. This step is trivial. We must only consider that writing to a file can cause an exception and that's why we need to change the logic for error handling slightly, opening and closing the flow of input and output to the file.

Here is what we finally get as a **complete source code of the program**:

HtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

class HtmlTagRemover
{
    private const string InputFileName = "Problem1.html";
    private const string OutputFileName = "Problem1.txt";
    private const string Charset = "windows-1251";

    static void Main()
    {
        if (!File.Exists(InputFileName))
        {
```

```
            Console.WriteLine(
                "File " + InputFileName + " not found.");
            return;
        }

        StreamReader reader = null;
        StreamWriter writer = null;
        try
        {
            Encoding encoding = Encoding.GetEncoding(Charset);
            reader = new StreamReader(InputFileName, encoding);
            writer = new StreamWriter(OutputFileName, false,
                encoding);

            string line;
            while ((line = reader.ReadLine()) != null)
            {
                line = RemoveAllTags(line);
                line = RemoveDoubleNewLines(line);
                line = TrimNewLines(line);
                if (!string.IsNullOrEmpty(line))
                {
                    writer.WriteLine(line);
                }
            }
        }
        catch (IOException)
        {
            Console.WriteLine(
                "Can not read file " + InputFileName + ".");
        }
        finally
        {
            if (reader != null)
            {
                reader.Close();
            }

            if (writer != null)
            {
                writer.Close();
            }
        }
    }
```

```
/// <summary>
/// Replaces every tag with new line
/// </summary>
private static string RemoveAllTags(string str)
{
   string strWithoutTags =
      Regex.Replace(str, "<[^>]*>", "\n");
   return strWithoutTags;
}

/// <summary>
/// Replaces sequence of new lines with only one new line
/// </summary>
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "[\n]+";
   return Regex.Replace(str, pattern, "\n");
}

/// <summary>
/// Removes new lines from start and end of string
/// </summary>
private static string TrimNewLines(string str)
{
   int start = 0;
   while (start < str.Length && str[start] == '\n')
   {
      start++;
   }

   int end = str.Length - 1;
   while (end >= 0 && str[end] == '\n')
   {
      end--;
   }

   if (start > end)
   {
      return string.Empty;
   }

   string trimmed = str.Substring(start, end - start + 1);
   return trimmed;
```

```
   }
}
```

Testing the Solution

Until now, we were testing the individual steps for the solution of the task. Through the tests of individual steps we reduced the possibility of errors, but that does not mean that we should not test the whole solution. We may have missed something, right? Now let's **thoroughly test the code**.

- Test with the **sample input** file from the problem statement. Everything works correctly.
- Test our **"complex" example**. Everything works fine.
- Test the **border cases** and run an output test.
- We test with a **blank file**. Output is correct – an empty file.
- Test with a file that contains **only one word** "**Hello**" and does not contain tags. The result is correct – the output contains only the word "**Hello**".
- Test with a **file that contains only tags and no** text. The result is again correct – an empty file.
- Try to **put blank lines** of at the most amazing places in the input file. These empty lines should all be removed. For example we can run the following test:

```
  Hello

<br><br>



<b>I<b> am here

I am not <b>here</b>
```

The result is as follows:

```
  Hello
I
 am here
I am not
Here
```

It seems we found a **small defect**. There is a **space at the beginning of some of the lines**.

Fixing the Leading Spaces Defect

Under the problem description it is not clear whether this is a defect but let's try to **fix it**. We could add the following code when processing the next line of the input file:

```
line = line.Trim();
```

The defect is fixed, but only from the first line. We run the debugger and we notice why it is so. The reason is that we print into the output file a string of characters with value "**I\n am here**" and so we get a space after a blank line. We can correct the defect, by replacing all blank lines, followed by white space (blank lines, spaces, tabs, etc.) with a single blank line. Here is the **correction**:

```
private static string RemoveDoubleNewLines(string str)
{
   string pattern = "\n\\s+";
   return Regex.Replace(str, pattern, "\n");
}
```

We fixed that error too. Now we have only to change this name to a more appropriate one, for example **RemoveNewLinesWithWhiteSpace(…)**.

Now we need to **test again after the "fixes"** in the code (**regression test**). We put new lines and spaces scattered randomly and make sure that everything works correctly now.

Performance Test

One last test remains: **performance**. We can create easily create a large input file. We open a site, for example http://www.microsoft.com, grab the source code and copy it 1000 times. We get a large enough input file. In our case, we get a **44 MB file** with 947,000 lines. Processing it takes under 10 seconds, which is a **perfectly acceptable speed**. When we test the solution we should not forget that the processing of the file depends on our hardware (our test was performed in 2009 on an average fast laptop).

Taking a look at the result, however, we notice a very troublesome problem. There are parts of a tag. More precisely, we see the following:

```
<!--
var s_pageName="home page"
//-->
```

It quickly becomes clear that we **missed a very interesting case**. In an HTML tag can be closed few lines after its opening, e.g. a **single tag may span several consecutive lines**. That was exactly our case: we have a

comment tag that contains JavaScript code. If the program worked correctly, it would have cut the entire tag rather than keep it in the source file.

Did you see how testing is useful and **how testing is important**? In some big companies (like Microsoft) having a solution without tests is considered as only 50% of the work. This means that if you write code for 2 hours, you should spend on testing (manual or automated) at least 2 more hours! This is the only way to create high-quality software.

What a pity that we discovered the problem just now, instead of at the beginning, when we were checking whether our idea for the task is correct, before we wrote the program. Sometimes it happens, unfortunately.

How to Fix the Problem with the Tag at Two Lines?

The first idea that occurs to us is to **load in memory the entire input file** and process it as one big string rather than row by row. This is an idea that seems to work but will run slow and **consume large amounts of memory**. Let's look for another idea.

A New Idea: Processing the Text Char by Char

Obviously **we cannot read the file line by line**. Can we **read it character by character**? If yes, how we will treat tags? It occurs to us that if we read the file character by character, we can know at any moment, whether we are in or outside of a tag, and if we are outside the tag, we can print everything that we read (followed by a new line). We need to avoid adding new lines, as well as and trailing whitespace. We will get something like this:

```
bool inTag = false;
while (! <end of file is reached>)
{
   char ch = (read the next character);
   if (ch == '<')
   {
      inTag = true;
   }
   else if (ch == '>')
   {
      inTag = false;
   }
   else
   {
      if (!inTag)
      {
         PrintBuffer(ch);
      }
   }
```

```
}
```

Implementing the New Idea

The idea is **very simple and easy to implement**. If we implement it directly, we will have a problem with empty lines and the problem of merging text from adjacent tags. To solve this problem, we can accumulate the text in the **StringBuilder** and print it at the end of file or when switching from text to a tag. We will get something like this:

```
bool inTag = false;
StringBuilder buffer = new StringBuilder();
while (! <end of file is reached>)
{
   char ch = (read the next character);
   if (ch == '<')
   {
      if (!inTag)
      {
         PrintBuffer(buffer);
      }
      buffer.Clear();
      inTag = true;
   }
   else if (ch == '>')
   {
      inTag = false;
   }
   else
   {
      if (!inTag)
      {
         buffer.Append(ch);
      }
   }
}
PrintBuffer(buffer);
```

The missing **PrintBuffer(…)** method should clean the whitespace from the text in the buffer and print it in the output followed by a new line. Exception is when we have whitespace only in the buffer (it should not be printed).

We already have most of the code, so **step-by-step implementation mat not be necessary**. We can just replace the pieces of wrong old code with the new code implementing the new idea. If we add the logic for avoiding empty

lines as well as reading input and writing the result we obtain is a **complete solution to the task with the new algorithm**:

SimpleHtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

public class SimpleHtmlTagRemover
{
   private const string InputFileName = "Problem1.html";
   private const string OutputFileName = "Problem1.txt";
   private const string Charset = "windows-1251";
   private static Regex regexWhitespace = new Regex("\n\\s+");

   static void Main()
   {
      if (!File.Exists(InputFileName))
      {
         Console.WriteLine(
            "File " + InputFileName + " not found.");
         return;
      }

      StreamReader reader = null;
      StreamWriter writer = null;
      try
      {
         Encoding encoding = Encoding.GetEncoding(Charset);
         reader = new StreamReader(InputFileName, encoding);
         writer = new StreamWriter(OutputFileName, false,
            encoding);
         RemoveHtmlTags(reader, writer);
      }
      catch (IOException)
      {
         Console.WriteLine(
            "Cannot read file " + InputFileName + ".");
      }
      finally
      {
         if (reader != null)
         {
```

```
            reader.Close();
        }
        if (writer != null)
        {
            writer.Close();
        }
    }
}

/// <summary>Removes the tags from a HTML text</summary>
/// <param name="reader">Input text</param>
/// <param name="writer">Output text (result)</param>
private static void RemoveHtmlTags(
    StreamReader reader, StreamWriter writer)
{
    StringBuilder buffer = new StringBuilder();
    bool inTag = false;
    while (true)
    {
        int nextChar = reader.Read();
        if (nextChar == -1)
        {
            // End of file reached
            PrintBuffer(writer, buffer);
            break;
        }
        char ch = (char)nextChar;
        if (ch == '<')
        {
            if (!inTag)
            {
                PrintBuffer(writer, buffer);
            }
            buffer.Clear();
            inTag = true;
        }
        else if (ch == '>')
        {
            inTag = false;
        }
        else
        {
            // We have other character (not "<" or ">")
            if (!inTag)
```

```
                {
                    buffer.Append(ch);
                }
            }
        }
    }

    /// <summary>Removes the whitespace and prints the buffer
    /// in a file</summary>
    /// <param name="writer">the result file</param>
    /// <param name="buffer">the input for processing</param>
    private static void PrintBuffer(
        StreamWriter writer, StringBuilder buffer)
    {
        string str = buffer.ToString();
        string trimmed = str.Trim();
        string textOnly = regexWhitespace.Replace(trimmed, "\n");
        if (!string.IsNullOrEmpty(textOnly))
        {
            writer.WriteLine(textOnly);
        }
    }
}
```

The input file is **read character by character** with the class **StreamReader**.

Originally the buffer for accumulating of text is empty. In the main loop we **analyze each read character**. We have the following cases:

- If we get to the **end of file**, we print whatever is in the buffer and the algorithm ends.
- When we encounter the character "**<**" (**start tag**) we first print the buffer (if we find that the transition is from text to tag). Then we clear the buffer and set **inTag** = **true**.
- When we encounter the character "**>**" (**end tag**) we set **inTag** = **false**. This will allow the next characters after the tag to accumulate in the buffer.
- When we encounter **another character** (text or blank space), it is added to the buffer, if we are outside tags. If we are in a tag the character is ignored.

Printing of the buffer takes care of **removing empty lines** in text and clearing the empty space at the beginning and end of text (trimming the leading and trailing whitespace). How exactly we do this, we already discussed in the previous solution of the problem.

In the second solution the processing of the buffer is much lighter and shorter, so the buffer is processed immediately before printing.

In the previous solution of the task we used regular expressions for replacing with the static methods of the class **Regex**. For **improved performance** now we create the regular expression object just once (as a **static** field). Thus the regular expression pattern is compiled just once to a state machine.

Testing the New Solution

It remains to **test thoroughly the new solution**. We have to perform all tests conducted on the previous solution. Add test with tags, which are spread over several lines. Again, test performance with the Microsoft website copied 1000 times. Assure that the program works correctly and is even faster.

Let's try with another site, such as the official website of this book – http://www.introprogramming.info (as of April 2011). Again, take the source code of the site and run the solution of our task with it. After carefully reviewing the input data (source code on the website of the book) and the output file, we notice that **there is a problem again**. Some content of this tag is printed in the output file:

```
<!--
<br />
<br />
Read the free book by Svetlin Nakov and team for developing with
Java.
…
…
-->
```

Where Is the Problem?

The problem seems to occur when **one tag meets another tag, before the first tag is closed**. This can happen in HTML comments. Here's how to get to the error:

```
<!--      1. InTag = true
<br />
<br />    2. InTag = false
…
```

As we know, in the solution of the task we use Boolean variable (**inTag**), to know whether the current character is in the tag or not. On the figure above we have shown that in moment 1 we set **inTag = true**. So far so good. Then comes moment 2, where the current character read is "**>**". At this point we find **inTag = false**. The problem is that the tag, which is open from moment 1 is not yet closed, and the Boolean variable indicates that we are not in the

tag anymore and the following characters are saved in the buffer. If between the two tags for a new line (**
**) we have text, it would also be saved in the buffer.

How to Fix the Problem?

It turned out that in the second solution **there is a mistake**. The program does not work correctly in the presence of **nested tags in a comment tag**. By Boolean variable can only know whether we are in a tag or not, but cannot remember if we are still in the preceding. This tells us that instead of using a Boolean variable, we can store the number of tags in which we are (in variable of type **int** – **tag counter**). We will modify the solution:

```
int openedTags = 0;
StringBuilder buffer = new StringBuilder();
while (! <end of file is reached>)
{
  char ch = (read the next character);
  if (ch == '<')
  {
    if (openedTags == 0)
    {
      PrintBuffer(buffer);
    }
    buffer.Remove(0, buffer.Length);
    openedTags++;
  }
  else if (ch == '>')
  {
    openedTags--;
  }
  else
  {
    if (openedTags == 0)
    {
      buffer.Append(ch);
    }
  }
}
PrintBuffer(buffer);
```

In the main loop we analyze each read character. We have the following cases:

- If we get to the **end of the file**, print whatever is in the buffer and the **algorithm ends**.

- When we encounter the character "**<**" (**start tag**) first we print the buffer (if we find that the transition from text to the tag). Then we clear the buffer and **increase the counter** by one.
- When we encounter the character "**>**" (**end tag**) we **reduce the counter** by one. Closing of a nested tag will not allow accumulation in the buffer. If after closing a tag we are out of all tags, the characters will begin to accumulate in the buffer.
- When we encounter **another character** (text or blank space), it is **added to the buffer**, if we are outside all tags. If we are inside a tag – the **character is ignored**.

It remains to **write the whole solution again and then test it**. The logic for reading the input file and printing the buffer remains the same:

SimpleHtmlTagRemover.cs

```
using System;
using System.IO;
using System.Text;
using System.Text.RegularExpressions;

public class SimpleHtmlTagRemover
{
  private const string InputFileName = "Problem1.html";
  private const string OutputFileName = "Problem1.txt";
  private const string Charset = "windows-1251";
  private static Regex regexWhitespace = new Regex("\n\\s+");

  static void Main()
  {
    if (!File.Exists(InputFileName))
    {
      Console.WriteLine(
        "File " + InputFileName + " not found.");
      return;
    }

    StreamReader reader = null;
    StreamWriter writer = null;
    try
    {
      Encoding encoding = Encoding.GetEncoding(Charset);
      reader = new StreamReader(InputFileName, encoding);
      writer = new StreamWriter(OutputFileName, false,
        encoding);
```

```
        RemoveHtmlTags(reader, writer);
    }
    catch (IOException)
    {
        Console.WriteLine(
            "Cannot read file " + InputFileName + ".");
    }
    finally
    {
        if (reader != null)
        {
            reader.Close();
        }
        if (writer != null)
        {
            writer.Close();
        }
    }
}

/// <summary>Removes the tags from a HTML text</summary>
/// <param name="reader">Input text</param>
/// <param name="writer">Output text (result)</param>
private static void RemoveHtmlTags(
    StreamReader reader, StreamWriter writer)
{
    int openedTags = 0;
    StringBuilder buffer = new StringBuilder();
    while (true)
    {
        int nextChar = reader.Read();
        if (nextChar == -1)
        {
            // End of file reached
            PrintBuffer(writer, buffer);
            break;
        }
        char ch = (char)nextChar;
        if (ch == '<')
        {
            if (openedTags == 0)
            {
                PrintBuffer(writer, buffer);
                buffer.Length = 0;
```

```
          }
          openedTags++;
        }
        else if (ch == '>')
        {
          openedTags--;
        }
        else
        {
          // We aren't in tags (not "<" or ">")
          if (openedTags == 0)
          {
            buffer.Append(ch);
          }
        }
      }
    }

    /// <summary>Removes the whitespace and prints the buffer
    /// in a file</summary>
    /// <param name="writer">the result file</param>
    /// <param name="buffer">the input for processing</param>
    private static void PrintBuffer(
      StreamWriter writer, StringBuilder buffer)
    {
      string str = buffer.ToString();
      string trimmed = str.Trim();
      string textOnly = regexWhitespace.Replace(trimmed, "\n");
      if (!string.IsNullOrEmpty(textOnly))
      {
        writer.WriteLine(textOnly);
      }
    }
}
```

Testing the New Solution

Again we **test the solution of the problem**. We **perform all tests** made on the previous solution (see section "Testing the Solution"). We also try the site of MSDN (http://msdn.microsoft.com). Let's carefully check the output file. We can see that at its end the file contains wrong characters (in April 2011). After carefully reviewing the source code of the MSDN site, we notice that there is an incorrect representation of the character "**>**" (to visualize this character in the HTML document "**>**" should be used, not "**>**"). However, this is an error in the MSDN site, not in our program.

Now it remains to **test the performance** of our program with the site of this book (http://www.introprogramming.info) copied 1000 times. We assure that the program works fast enough for it too.

Finally we are **ready for the next task**.

Problem 2: Escape from Labyrinth

We are given a **labyrinth**, which consists of **N x N squares** and each of it can be **passable (0) or not (x)**. Our hero Jack is in one of the squares (*****):

x	x	x	x	x	x
0	x	0	0	0	x
x	*	0	x	0	x
x	x	x	x	0	x
0	0	0	0	0	x
0	x	x	x	0	x

Two of the squares are **neighboring**, if they have a common wall. In one step Jack can pass from one passable square to its neighboring passable square. If Jack steps in a cell, which is on the edge of the labyrinth, he can go out from the labyrinth with one step.

Write a program, which by a given labyrinth prints the **minimal number of steps**, which Jack needs, **to go out from the labyrinth** or **-1** if there is no way out.

The input data is read from a text file named **Problem2.in**. On the first line of the file is the number N (2 < N < 100). On the each of next N lines there are N characters, each of them is either "0" or "x" or "*". The output is one number and must be in the file **Problem2.out**.

Sample input – **Problem2.in**:

```
6
xxxxxx
0x000x
x*0x0x
xxxx0x
00000x
0xxx0x
```

Sample output – **Problem2.out**:

```
9
```

Figure Out an Idea for a Solution

We have a labyrinth and we should **find the shortest path** in it. This is not an easy task and we should think a lot or we should read somewhere how to solve such kinds of tasks.

Our algorithm will begin its movement from the initial point we are given. We know we can move to a neighboring cell horizontally or vertically, but not diagonally. Our algorithm must traverse the labyrinth in some way, to find the shortest path in it. How to traverse the cells in the labyrinth?

One possible decision is the following: we start from the initial cell. Move to one of its neighboring cells, after this in a neighboring cell of the current (which is passable and still unvisited), after this in a neighboring cell of the last visited (which is passable and still unvisited) and we go on forward recursively until we reach an exit of the labyrinth, or we reach a place where we can't continue (there is no neighboring cell which is free or unvisited). In this moment we go back from the recursion (to the previous cell) and visit another neighboring cell for the previous cell. If we can't continue, we go back again. The described **recursive process** is the process of traversing the labyrinth **in depth** (remember the chapter "Recursion" and DFS traversal).

The question "Is it needed to walk through one cell more than once" occurs to us? If we walk through one cell at most once, we can walk through the whole labyrinth faster and if there is an exit, we will find it. But **will this be the minimal path?** If we draw the whole process on a paper, we will find out quickly the path will not be the minimal.

If we mark the cell we leave on the way back of the recursion as free, this will allow us to reach each cell repeatedly, coming from a different path. The **full recursive walk of the labyrinth will find all possible paths** from the initial cell to any other cell. From all the found paths we can **choose the shortest path** to a cell on the bound of the labyrinth (exit) and that's how we will find a solution for the problem.

Verification of the Idea

It seems we have an idea for solving the problem: with recursive walk we **find all the possible paths** in the labyrinth from the initial cell to a cell on the bounds of the labyrinth and from all these paths we **choose the shortest one**. Let's check the idea.

We take a sheet of paper and make one example of the labyrinth. We try the algorithm. It's obvious it finds all the paths from the initial cell to the one of the exits and it travels a lot forwards-backwards. As a result it finds all exits and among all paths it can be chosen the shortest one.

Does the idea work **if there is no exit**? We create a second labyrinth, which is without exit. We try out the algorithm on it, again on a sheet of paper. We see after long circulation forwards-backwards that the algorithm does not find an exit and finishes.

It looks we have a correct idea for solving the problem. Let's move forward and think for the data structures.

What Data Structures to Use?

First, we have to decide **how to store the labyrinth**. It's natural to use a matrix of characters, just as the one on the figure. We will consider that one cell is passable and we can enter it, if it has a character, different from the character '**x**'. We can store the labyrinth in a matrix of numbers and Boolean values, but the difference is not significant. The matrix of characters is comfortable for printing, and this will help us while debugging. There are not many options. We will store the labyrinth in a **matrix of characters**.

After this, we have to decide in what structure to **keep the visited** through the recursion (current path) **cells**. We always need the last visited cell. This leads us to a structure, which is "last in, first out", i.e. **stack**. We can use **`Stack<Cell>`**, where **`Cell`** is a class, containing the coordinates of one cell (number of row and number of column). It remains to think where to keep the found paths, to find the shortest of them. If we think of it, it is not necessary to keep all the paths. It is enough to keep the current path and the shortest till this moment. It's not even necessary to keep the shortest path till this moment but only its **length**. Every time we find a path to an exit of the labyrinth we can take its length and if it is shorter than the shortest path to this moment to keep it.

It seems we found **efficient data structures**. According to our recommendations for problem solving, it is early to write the code of the program, because we have to think of the efficiency of the algorithm.

Think About the Efficiency

Let's check our idea against **efficiency**. What are we doing? We find all the possible paths and we take the shortest. There's no argument the algorithm will not work, but if the labyrinth is way bigger, will it work fast?

To answer this question, we should think **how much paths there are**. If we take an empty labyrinth, on the each step of the recursion we will have an average number of 3 free cells to go (without the cell we are coming from).

If we have for example a labyrinth 10x10, the path could be 100 cells and while we travel on each step we will have 3 neighboring cells. It seems the numbers of paths are sort of 3 to the power of 100. It's obvious **the algorithm will slow down the computer very much and very fast**.

We found a serious problem with the algorithm. It **will work very slowly**, even with small labyrinths, and with bigger ones it will not work at all! The good news is that we haven't written a single line of code and the general change of our approach to the problem will not cost us much time.

Think of Another Idea

We found that **walking through all the paths in the labyrinth is wrong approach**, so we have to think of another.

Let's start with the initial cell and walk through all its neighboring cells and mark them as visited. **For each visited cell we can keep a number equal to the number of cells, which we have travelled to reach it** (the length of the minimal path from the initial cell to the current cell).

For the initial cell the length of the path is 0. For its neighboring cells it should be 1, because we can reach them from the initial cell with one move. For the neighboring cells for the neighbors of the initial cell the length of the path is 2. We can continue this way and we will get to **the following algorithm**:

1. Write the length of the path 0 for the initial cell. Mark it as visited.
2. For each neighboring cell to the initial we mark the length of the path is 1. Mark these cells as visited.
3. For each cell, which is, neighboring to a cell with length of the path 1 and it is not visited, write the length of the path is 2. Mark the cells as visited.
4. Continuing analogous, on the N step we find all the still unvisited cells, which are on a distance of N moves from the initial cell and mark them as visited.

Check the New Idea

To check whether the new idea for solving the "Escape from the Labyrinth" problem is correct we can **visualize the process.** We take another labyrinth to test our idea in a better way. At each step **k** our goal is to fill with the number **k** all cells that can be reached in **k** steps. If at step 0 we fill the initial cell with 0, at step 1 we fill all cells reachable in 1 step from the initial cell, at step 2 we fill all cells reachable in 2 steps, etc. we will be sure that when we fill a cell with a number, this number reflects **the minimal number of steps to reach this cell** starting from the initial cell, right?

Step 0 – mark the distance from the initial cell to itself with **0** (mark the free cells with "**-**"):

x	x	x	x	x	x
-	x	-	-	-	x
x	0	-	x	-	x
x	-	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Step 1– mark with **1** all the neighbors to cells with a value of 0:

x	x	x	x	x	x
-	x	-	-	-	x
x	0	1	x	-	x
x	1	-	x	-	x
x	-	-	-	-	x
-	x	x	x	-	x

Step 2 – mark with **2** all the passable neighbors to cells with value 1:

x	x	x	x	x	x
-	x	2	-	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	-	-	-	x
-	x	x	x	-	x

Step 3 – mark with 3 all passable neighbors to cells with value of 2:

x	x	x	x	x	x
-	x	2	3	-	x
x	0	1	x	-	x
x	1	2	x	-	x
x	2	3	-	-	x
-	x	x	x	-	x

Continuing this way, in a moment either we will reach a cell at the edge of the labyrinth (an exit) or we will find such a cell is unreachable. It seems like **our algorithm works correctly**. It will either find an exit or will find that there is no reachable exit. If at some step an exit is found, the path to it will be guaranteed to be **the shortest possible** (otherwise the exit should already be found at some of the earlier steps).

Breaking the Problem into Subproblems

Having invented the idea for solving the labyrinth escaping problem, it will be easy to break it into subproblems. The main subproblems could be: **reading the input** labyrinth, **finding the shortest path** to some of its exits and **printing the results**. The **path finding subproblem** could be further divided into subproblems (steps) which we discussed in the previous section.

Checking the Performance of the New Algorithm

Because we never visit a cell more than once, the number of steps, which this algorithm does, **should not be big**. For example, if we have a labyrinth with size 100 x 100, it will have 10,000 cells, we will visit each of the cells at most once and for each of them we will check every neighbor if it is free, i.e. we will check 4 times each cell. At the end we will do at most 40,000 checks and we will visit at most 10,000 cells. We will do a total amount of 50,000 operations. This means **the algorithm will work instantly**.

Check If the New Algorithm Is Correct

It seems this time we don't have a problem with the performance. We have a **fast algorithm**.

Let's check if it is correct. For this purpose we draw a bigger and more complex example on a sheet of paper, which has many exits and a lot of paths, and we begin to perform the algorithm. After this we try with a labyrinth with no exit. It seems the algorithm ends, but does not find an exit so it's working. We try another 2-3 examples and **convince ourselves this algorithm always finds the shortest path to an exit and always works fast**, because it visits each of the cells of the labyrinth at most once.

What Data Structures to Use?

With the new algorithm we walk consequently through all neighboring cells to the initial cell. We can put them into a **data structure**, for example in an array or better a list (or list of lists), because we can't add in the array.

Then we take the **list of the reached cells on the last step** and we add their neighbors in another list.

That's how if we index the lists we have **$list_0$**, which contains **the initial cell**, **$list_1$**, which contains passable **neighboring cells to the initial**, after this **$list_2$**, which contains **passable neighbors to $list_1$** and so on. At the N step we have the **$list_n$**, which contains all the cells, which we can **reach in exactly N steps**, i.e. which are **at a distance of n** from the initial cell.

It seems we can use a list of lists, to keep the cells on each step. If we think about it, to get the n list, we need the (**n-1**)-list. So it seems we don't need list of lists but **only the list from the last step**.

We can make general conclusion: cells are processed in the order of entry: when the cells of step **k** are finished, then we process the cells from step **k+1**, and just after them – the cells from step **k+2**, and so on. The process seems like a **queue**: earlier accessed cells are processed earlier. If we dig a bit inside, we will conclude, that we have just **re-invented the Breadth-First-Search algorithm** (read about BFS in Wikipedia).

To implement the BFS algorithm we can use a **queue of cells**. For this purpose we have to **define class `Cell`**, which contains the coordinates of

given cell (row and column). We can **keep the distance** from each cell to the initial cell in a matrix. If the distance is not calculated yet, we store -1.

If we think a little more, the distance from the initial cell can be kept in the cell itself (in the class **Cell**) instead of creating a special matrix for the distances. That way we will save memory.

Now we are clear about the data structures. Now we have to implement the algorithm step by step.

Step 1 – The Class Cell

We can begin with the **definition of the Cell class**. We need it to save the initial cell, from which begins the searching of the path. We will use auto-implemented properties to make the code shorter and more readable. Here is the **Cell** class:

```
public class Cell
{
  public int Row { get; set; }
  public int Column { get; set; }
  public int Distance { get; set; }
}
```

We can add a **constructor** to simplify the way we use this class:

```
public Cell(int row, int column, int distance)
{
  this.Row = row;
  this.Column = column;
  this.Distance = distance;
}
```

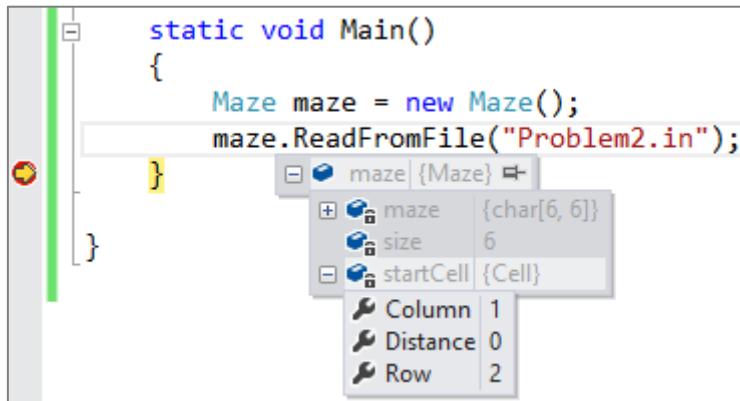
Generally it is a good idea **to test the code after each step**, but the above code is too simple to be tested. We will test is later as part of some more complex piece of code.

Step 2 – Reading the Input File

We will read the input file **line by line** using the well-known class **StreamReader**. On the each of the lines we will analyze the characters and we will write them in a matrix of characters. When we reach the character "*****" we will keep its coordinates in an instance of class **Cell** to know where to start the searching of the shortest path for getting out of the labyrinth.

We can **define a class Maze** and keep the matrix of the labyrinth and the initial cell in it:

Maze.cs

```
public class Maze
{
  private char[,] maze;
  private int size;
  private Cell startCell = null;

  public void ReadFromFile(string fileName)
  {
    using (StreamReader reader = new StreamReader(fileName))
    {
      // Read the maze size and create the maze
      this.size = int.Parse(reader.ReadLine());
      this.maze = new char[this.size, this.size];

      // Read the maze cells from the file
      for (int row = 0; row < this.size; row++)
      {
        string line = reader.ReadLine();
        for (int col = 0; col < this.size; col++)
        {
          this.maze[row, col] = line[col];
          if (line[col] == '*')
          {
            this.startCell = new Cell(row, col, 0);
          }
        }
      }
    }
  }
}
```

For simplicity we will skip processing the errors while reading and writing in a file. When an **exception occurs** we will skip to catch it in the main method and thus we will leave the CLR to print it on the console.

Testing the Input File Reading Code

We already have the class **Maze** and appropriate representation of data of the input file. To be sure the written so far is correct **we should test**. We can check if the matrix is truly filled as we print it on the console. The other possibility is to view the values of the fields in the class **Maze** through the debugger of Visual Studio. We add a **Main()** method which invokes the maze reading method and we test it:

```
static void Main()
{
   Maze maze = new Maze();
   maze.ReadFromFile("Problem2.in");
}
```

Through the Visual Studio debugger we get convinced that **the input file is correctly read** from the input file:

```
static void Main()
{
    Maze maze = new Maze();
    maze.ReadFromFile("Problem2.in");
}
```


Step 3 – Finding the Shortest Path

We can **implement the algorithm** directly from what we already discussed. We must define a queue and put in its beginning the initial cell. Afterwards we must take the cell in turn from the queue and add all of its passable unvisited neighbors in a loop. At each step there is a chance to enter in a cell, which is at the border of the labyrinth, and we see we have found an exit and the searching ends. We repeat the loop until the queue is empty. At each visitation of a given cell we check if it is free and if it is, we mark it as impassable. This way we avoid repeatedly visiting the same cell.

Here is how **the implementation of the algorithm** looks like:

```
public int FindShortestPath()
{
   // Queue for traversing the cells in the maze
   Queue<Cell> visitedCells = new Queue<Cell>();
   VisitCell(visitedCells, this.startCell.Row,
      this.startCell.Column, 0);

   // Perform Breath-First-Search (BFS)
   while (visitedCells.Count > 0)
   {
      Cell currentCell = visitedCells.Dequeue();
      int row = currentCell.Row;
```

```
        int column = currentCell.Column;
        int distance = currentCell.Distance;
        if ((row == 0) || (row == size - 1)
            || (column == 0) || (column == size - 1))
        {
            // We are at the maze border
            return distance + 1;
        }
        VisitCell(visitedCells, row, column + 1, distance + 1);
        VisitCell(visitedCells, row, column - 1, distance + 1);
        VisitCell(visitedCells, row + 1, column, distance + 1);
        VisitCell(visitedCells, row - 1, column, distance + 1);
    }

    // We didn't reach any cell at the maze border -> no path
    return -1;
}

private void VisitCell(Queue<Cell> visitedCells,
    int row, int column, int distance)
{
    if (this.maze[row, column] != 'x')
    {
        // The cell is free --> visit it
        maze[row, column] = 'x';
        Cell cell = new Cell(row, column, distance);
        visitedCells.Enqueue(cell);
    }
}
```

Checking after Step 3

Before the next step, **we must test, to check our algorithm**. We must try the **normal case** and the **border cases**, when there is no exit, when we step on an exit, when the input file doesn't exist or the square matrix is with size of 0. Only then can we start doing the next step. Let's start with **testing the normal (typical) case**. We create the following code to quickly test it:

```
static void Main()
{
    Maze maze = new Maze();
    maze.ReadFromFile("Problem2.in");
    Console.WriteLine(maze.FindShortestPath());
}
```

We run the above code over the sample input file from the problem description and **it works**. The code correctly returns the length of the shortest path to the nearest exit:

```
9
```

Now let's test the **border cases**, e.g. a **labyrinth of size 0**. Unfortunately we get the following result:

```
Unhandled Exception: System.NullReferenceException: Object
reference not set to an instance of an object.
 at Maze.FindShortestPath()
```

We've made a **mistake**. The problem is when the variable, in which we keep the **initial cell**, is initialized with **null**. This can happen in many scenarios. If the labyrinth has no cells (e.g. size of 0) or the initial cell is missing, the result that the program **should return is -1, but not an exception**.

To **fix the bug** we just found we can add a check in the beginning of the **FindShortestPath()** method:

```
public int FindShortestPath()
{
   if (this.startCell == null)
   {
      // Start cell is missing -> no path
      return -1;
   }
   …
```

We **retest the code** with the typical and the border cases. After the fix **it seems the algorithm works correctly** now.

Step 4 – Writing the Result to a File

It remains to write the result of the **FindShortestPath()** to the output file. This is a trivial problem:

```
public void SaveResult(String fileName, int result)
{
   using (StreamWriter writer = new StreamWriter(fileName))
   {
      writer.WriteLine("The shortest way is: " + result);
   }
}
```

Here is how **the complete source code of the solution** looks:

Maze.cs

```
using System;
using System.IO;
using System.Collections.Generic;

public class Maze
{
  private const string InputFileName = "Problem2.in";
  private const string OutputFileName = "Problem2.out";

  public class Cell
  {
    public int Row { get; set; }
    public int Column { get; set; }
    public int Distance { get; set; }

    public Cell(int row, int column, int distance)
    {
      this.Row = row;
      this.Column = column;
      this.Distance = distance;
    }
  }

  private char[,] maze;
  private int size;
  private Cell startCell = null;

  public void ReadFromFile(string fileName)
  {
    using (StreamReader reader = new StreamReader(fileName))
    {
      // Read maze size and create maze
      this.size = int.Parse(reader.ReadLine());
      this.maze = new char[this.size, this.size];

      // Read the maze cells from the file
      for (int row = 0; row < this.size; row++)
      {
        string line = reader.ReadLine();
        for (int col = 0; col < this.size; col++)
        {
          this.maze[row, col] = line[col];
```

```
                if (line[col] == '*')
                {
                    this.startCell = new Cell(row, col, 0);
                }
            }
        }
    }
}

public int FindShortestPath()
{
    if (this.startCell == null)
    {
        // Start cell is missing -> no path
        return -1;
    }

    // Queue for traversing the cells in the maze
    Queue<Cell> visitedCells = new Queue<Cell>();
    VisitCell(visitedCells, this.startCell.Row,
        this.startCell.Column, 0);

    // Perform Breath-First-Search (BFS)
    while (visitedCells.Count > 0)
    {
        Cell currentCell = visitedCells.Dequeue();
        int row = currentCell.Row;
        int column = currentCell.Column;
        int distance = currentCell.Distance;
        if ((row == 0) || (row == size - 1)
            || (column == 0) || (column == size - 1))
        {
            // We are at the maze border
            return distance + 1;
        }

        VisitCell(visitedCells, row, column + 1, distance + 1);
        VisitCell(visitedCells, row, column - 1, distance + 1);
        VisitCell(visitedCells, row + 1, column, distance + 1);
        VisitCell(visitedCells, row - 1, column, distance + 1);
    }

    // We didn't reach any cell at the maze border -> no path
    return -1;
```

```
    }

    private void VisitCell(Queue<Cell> visitedCells,
        int row, int column, int distance)
    {
        if (this.maze[row, column] != 'x')
        {
            // The cell is free --> visit it
            maze[row, column] = 'x';
            Cell cell = new Cell(row, column, distance);
            visitedCells.Enqueue(cell);
        }
    }

    public void SaveResult(string fileName, int result)
    {
        using (StreamWriter writer = new StreamWriter(fileName))
        {
            writer.WriteLine(result);
        }
    }

    static void Main()
    {
        Maze maze = new Maze();
        maze.ReadFromFile(InputFileName);
        int pathLength = maze.FindShortestPath();
        maze.SaveResult(OutputFileName, pathLength);
    }
}
```

Testing the Complete Solution of the Problem

After we have a solution of the problem **we must test it**. We have already tested **the typical case** and **the border cases** (**like missing exit** or **when the initial position stays at the labyrinth edge**). We will execute these tests again to get convinced that the algorithm behaves correctly:

Input	Output	Input	Output	Input	Output	Input	Output
0	-1	2 00 xx	-1	3 0x0 x*x 0x0	-1	3 000 000 00*	1

The algorithm works correctly. The output for each of the test is correct.

It remains to **test with a large labyrinth** (performance test), for example 1000 x 1000. We can make such a labyrinth very easy – with copy / paste. We perform the test and we convince ourselves the program is **working correctly for the big test** and works extremely fast – **there is no delay**.

While testing **we should try every way to break our solution**. We run a few **more difficult examples** (for example a labyrinth with passable cells in the form of **spiral**). We can put large labyrinth with a lot of paths, but without exit. We can try whatever else we wish.

At the end we make sure, that **we have a correct solution** and we pass to the next problem from the exam.

Problem 3: Store for Car Parts

A company is planning to create a system for **managing a store for auto parts**. A single **part** can be used for different car models and it has following characteristics: **code**, **name**, **category** (e.g. suspension, tires and wheels, engine, accessories and etc.), **purchase price**, **sale price**, **list of car models**, with which it is compatible (each car is described with brand, model and year of manufacture, e. g. Mercedes C320, 2008) and **manufacturing company**. **Manufacturing companies** are described with name, country, address, phone and fax.

Design a set of classes with relationships between them, which model the data for the store. Write a **demonstration program**, which demonstrates the classes and their all functionality work correctly with some sample data.

Inventing an Idea for Solution

We have a non-algorithmic problem which is intended to check whether the students at the exam know how to use **object-oriented programming** (OOP), how to design classes and relationships between them to model real-world objects (**object-oriented analysis and design**) and how to use appropriate **data structures** to hold collections of objects.

We are required to create an aggregation of classes and relationships between them, which have to describe the data of the store. We have to find **which nouns are important** for solving the problem. They are objects from the real world, which correspond to **classes**.

Which are these nouns that interest us? We have a **store**, **car parts**, **cars** and **manufacturing companies**. We have to create a class defining a store. It could be named `Shop`. Other classes are `Part`, `Car` and `Manufacturer`. In the requirements of the problem there are other nouns too, like code for one part or year of manufacturing of given car. For these nouns we are not creating individual classes, but instead these will be fields in the already created classes. For example in the `Part` class there will be let's say a field `code` of `string` type.

We already know **which will be our classes, and fields to describe them**. We have to identify the relationships between the objects.

Checking the Idea

We **will not check the idea** because there is nothing to be proven with examples and counterexamples or checked whether it will work. We need to write few classes to model a real-world situation: a store for car parts.

What Data Structures to Use to Describe the Relationship between Two Classes?

The data structures, needed for this problem, are of two main groups: **classes** and **relationships between the classes**. The interesting part is how to describe relationships.

To **describe a relationship** (link) between two classes we can use an **array**. With an array we have access of its elements by index, but once it is created we can't change its length. This makes it **uncomfortable for our problem**, because we don't know how many parts we will have in the store and more parts can be delivered or somebody can buy parts so we have to delete or change the data. **`List<T>` is more comfortable**. It has the advantages of an array and also is with variable length and it is easy to add or delete elements.

So far it seems **`List<T>`** is the **most appropriate** for holding aggregations of objects inside another object. To be convinced we will analyze a few more data structures. For example **hash-table** – it is not appropriate in this case, because the structure "parts" is not of the key-value type. It would be appropriate if each of the parts in the store has unique number (e.g. barcode) and we needed to search them by this unique number. Structures like **stack** and **queue** are inappropriate.

The structure "**set**" and its implementation **`HashSet<T>`** is used when we have **uniqueness** for given key. It would be good sometimes to use this structure to avoid duplicates. We must recall that **`HashSet<T>`** requires the methods **`GetHashCode()`** and **`Equals(…)`** to be correctly defined by the **`T`** type.

Our final decision is to use **`List<T>`** for the aggregations and **`HashSet<T>`** for the aggregations which require uniqueness.

Dividing the Task into Subtasks

Now we have to think from where to start writing the code. If we start to write the **`Shop`** class, we will need the **`Part`** class. This reminds us we will have to start with a class, which does not depend on others. We will divide the writing of each class to a **subtask**, and we will start from the independent classes:

- Class describing a car – **`Car`**
- Class describing manufacturer of parts – **`Manufacturer`**
- Class or enumeration for the categories of the parts – **`PartCategory`**

- Class describing part for a car – **Part**
- Class for the store – **Shop**
- Class for testing rest of the classes with sample data – **TestShop**

Implementation: Step by Step

We start writing classes, which we described in our idea. We will create them in the same sequence as in the list above.

Step 1: The Class Car

We start solving the problem by **defining the class Car**. In the definition we have three fields, which keep the manufacturer, the model and the year of manufacturing of the car and the standard method **ToString()**, which returns a human-readable string holding the information about the car. We define the class **Car** in the following way:

Car.cs

```
public class Car
{
  private string brand;
  private string model;
  private int productionYear;

  public Car(string brand, string model, int productionYear)
  {
    this.brand = brand;
    this.model = model;
    this.productionYear = productionYear;
  }

  public override string ToString()
  {
    return "<" + this.brand + "," + this.model + ","
      + this.productionYear + ">";
  }
}
```

Note that the class **Car** is designed to be **immutable**. This means that once created, the car's properties cannot be later modified. This design is not always the best choice. Sometimes we want the class properties to be freely modifiable; sometimes. For our case the immutable design will work well.

Testing the Class Car

Once we have the class **Car**, we could test it by the following code:

```
Car bmw316i = new Car("BMW", "316i", 1994);
Console.WriteLine(bmw316i);
```

The result is as expected:

```
<BMW,316i,1994>
```

We are convinced the class **Car** is correct so far and we can continue with the other classes.

Step 2: The Class Manufacturer

We have to implement the **definition of the class Manufacturer**, which describes the manufacturer for given part. It will have five fields – name, country, address, phone number and fax. The class will be **immutable**, because we will not need to change its members after creation. We also define the standard method **ToString()** for representing the object as human-readable string.

Manufacturer.cs

```
public class Manufacturer
{
   private string name;
   private string country;
   private string address;
   private string phoneNumber;
   private string fax;

   public Manufacturer(string name, string country,
      string address, string phoneNumber, string fax)
   {
      this.name = name;
      this.country = country;
      this.address = address;
      this.phoneNumber = phoneNumber;
      this.fax = fax;
   }

   public override string ToString()
   {
      return this.name + " <" + this.country + "," + this.address
         + "," + this.phoneNumber + "," + this.fax + ">";
   }
}
```

Testing the Class Manufacturer

We test the class **Manufacturer** just like we tested the class **Car**. It works.

Step 3: The Part Category Enumeration

Part categories are **fixes set of values** and do not have additional details (like name, code and description). This makes them perfect to be modeled as **enumeration**:

PartCategory.cs

```
public enum PartCategory
{
   Engine,
   Tires,
   Exhaust,
   Suspention,
   Brakes
}
```

Step 4: The Class Part

Now we have to **define the class Part**. Its definition will include the following fields: name, code, category, list with cars, where we can use the given part, starting and closing price and manufacturer. Here we will use the data structure **HashSet<Car>** to hold all compatible cars.

The field that keeps the manufacturer of the part will be of **Manufacturer** class, because the task requires us to keep additional information about the manufacturer. If it was required to keep only the name of the manufacturer (as in the case with class **Car**) this class should not be necessary. We would have a field of **string** type.

We need a method for adding a car (object of type **Car**) to the list of cars (in **HashSet<Car>**). It will be named **AddSupportedCar(Car car)**.

Below is the code of the class **Part** which is also designed as set of **immutable fields** (except that it accepts adding cars):

Part.cs

```
public class Part
{
   private string name;
   private string code;
   private PartCategory category;
   private HashSet<Car> supportedCars;
   private decimal buyPrice;
```

```
    private decimal sellPrice;
    private Manufacturer manufacturer;

    public Part(string name, decimal buyPrice, decimal sellPrice,
        Manufacturer manufacturer, string code,
        PartCategory category)
    {
        this.name = name;
        this.buyPrice = buyPrice;
        this.sellPrice = sellPrice;
        this.manufacturer = manufacturer;
        this.code = code;
        this.category = category;
        this.supportedCars = new HashSet<Car>();
    }

    public void AddSupportedCar(Car car)
    {
        this.supportedCars.Add(car);
    }

    public override string ToString()
    {
        StringBuilder result = new StringBuilder();
        result.Append("Part: " + this.name + "\n");
        result.Append("-code: " + this.code + "\n");
        result.Append("-category: " + this.category + "\n");
        result.Append("-buyPrice: " + this.buyPrice + "\n");
        result.Append("-sellPrice: " + this.sellPrice + "\n");
        result.Append("-manufacturer: " + this.manufacturer +"\n");
        result.Append("---Supported cars---" + "\n");
        foreach (Car car in this.supportedCars)
        {
            result.Append(car);
            result.Append("\n");
        }
        result.Append("----------------------\n");
        return result.ToString();
    }
}
```

In the class **Part** we use **HashSet<Car>** so it is necessary to **redefine the methods Equals(…) and GetHashCode() for the class Car**:

```
// The Equals(…) and GetHashCode() methods for the class Car

public override bool Equals(object obj)
{
   Car otherCar = obj as Car;
   if (otherCar == null)
   {
      return false;
   }
   bool equals =
      object.Equals(this.brand, otherCar.brand) &&
      object.Equals(this.model, otherCar.model) &&
      object.Equals(this.productionYear,otherCar.productionYear);
   return equals;
}

public override int GetHashCode()
{
   const int prime = 31;
   int result = 1;
   result = prime * result + ((this.brand == null) ? 0 :
      this.brand.GetHashCode());
   result = prime * result + ((this.model == null) ? 0 :
      this.model.GetHashCode());
   result = prime * result + this.productionYear;
   return result;
}
```

Testing the Class Part

We test the class **Part**. It is a bit more complicated than when testing the classes **Car** and **Manufacturer**, because **Part** it is more complex class. We can create a part, assign all its properties and print it:

```
Manufacturer bmw = new Manufacturer("BWM",
   "Germany", "Bavaria", "665544", "876666");
Part partEngineOil = new Part("BMW Engine Oil",
   633.17m, 670.0m, bmw, "Oil431", PartCategory.Engine);
Car bmw316i = new Car("BMW", "316i", 1994);
partEngineOil.AddSupportedCar(bmw316i);
Car mazdaMX5 = new Car("Mazda", "MX5", 1999);
partEngineOil.AddSupportedCar(mazdaMX5);
Console.WriteLine(partEngineOil);
```

Seems like the result is **correct**:

```
Part: BMW Engine Oil
-code: Oil431
-category: Engine
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
<Mazda,MX5,1999>
----------------------
```

Before we can continue with the next class, we could **test for duplicated cars** in the set of supported cars for certain part. Duplicates are not allowed by design and we should check whether this is enforced:

```
Manufacturer bmw = new Manufacturer("BWM",
  "Germany", "Bavaria", "665544", "876666");
Part partEngineOil = new Part("BMW Engine Oil",
  633.17m, 670.0m, bmw, "Oil431", PartCategory.Engine);
partEngineOil.AddSupportedCar(new Car("BMW", "316i", 1994));
partEngineOil.AddSupportedCar(new Car("BMW", "X5", 2006));
partEngineOil.AddSupportedCar(new Car("BMW", "X5", 2007));
partEngineOil.AddSupportedCar(new Car("BMW", "X5", 2006));
partEngineOil.AddSupportedCar(new Car("BMW", "316i", 1994));
Console.WriteLine(partEngineOil);
```

The result is **correct**. The duplicated cars are taken into account only once:

```
Part: BMW Engine Oil
-code: Oil431
-category: Engine
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
<BMW,X5,2006>
<BMW,X5,2007>
----------------------
```

Step 5: The Class Shop

We already have all needed classes for **creating the class Shop**. It will have two fields: name and list of parts, which are for sale. The list will be **List<Part>**. We will add the method **AddPart(Part part)**, with which we

will add new parts. With a redefined **ToString()** we will print the name of the shop and the parts in it.

Here is an example of implementation of our class **Shop** holding the catalog of auto parts (its name is immutable but it can add parts):

Shop.cs

```
public class Shop
{
   private string name;
   private List<Part> parts;

   public Shop(string name)
   {
      this.name = name;
      this.parts = new List<Part>();
   }

   public void AddPart(Part part)
   {
      this.parts.Add(part);
   }

   public override string ToString()
   {
      StringBuilder result = new StringBuilder();
      result.Append("Shop: " + this.name + "\n\n");
      foreach (Part part in this.parts)
      {
         result.Append(part);
         result.Append("\n");
      }
      return result.ToString();
   }
}
```

It might be a **subject of discussion whether we should use List<Part> or Set<Part>** for the parts in the car shop. The **set data structure** has an advantage that it **avoids any duplicates**. Thus if we have for example few tires of certain model, they will be found only once in the set. To use set we need to be sure the parts are uniquely identified by their code or by some other unique identifier. In our case we assume we could have parts with exactly the same code, name, etc. which come at different buy and sell prices (e.g. if the prices change over the time). So we need to allow duplicated parts

and thus **using a set will not be appropriate**. Parts in the shop will be kept in **List<Part>**.

We will test the class **Shop** though the especially written class **TestShop**.

Step 6: The Class TestShop

We created all classes we need. We have to create one more, with which we will have **to demonstrate the usage of the rest of the classes**. It will be named **TestShop**. In the **Main()** method we will create two manufacturers and a few cars. We will add them to two parts. We will add the parts to the **Shop**. At the end we will print everything on the console.

TestShop.cs

```
public class TestShop
{
  static void Main()
  {
    Manufacturer bmw = new Manufacturer("BWM",
      "Germany", "Bavaria", "665544", "876666");
    Manufacturer lada = new Manufacturer("Lada",
      "Russia", "Moscow", "653443", "893321");

    Car bmw316i = new Car("BMW", "316i", 1994);
    Car ladaSamara = new Car("Lada", "Samara", 1987);
    Car mazdaMX5 = new Car("Mazda", "MX5", 1999);
    Car mercedesC500 = new Car("Mercedes", "C500", 2008);
    Car trabant = new Car("Trabant", "super", 1966);
    Car opelAstra = new Car("Opel", "Astra", 1997);

    Part cheapPart = new Part("Tires 165/50/R13", 302.36m,
      345.58m, lada, "T332", PartCategory.Tires);
    cheapPart.AddSupportedCar(ladaSamara);
    cheapPart.AddSupportedCar(trabant);

    Part expensivePart = new Part("Universal Car Engine",
      6733.17m, 6800.0m, bmw, "EU33",  PartCategory.Engine);
    expensivePart.AddSupportedCar(bmw316i);
    expensivePart.AddSupportedCar(mazdaMX5);
    expensivePart.AddSupportedCar(mercedesC500);
    expensivePart.AddSupportedCar(opelAstra);

    Shop newShop = new Shop("Tuning Pro Shop");
    newShop.AddPart(cheapPart);
    newShop.AddPart(expensivePart);
```

```
        Console.WriteLine(newShop);
    }
}
```

This is **the result** of the execution of the above code:

```
Shop: Tuning Pro Shop

Part: Tires 165/50/R13
-code: T332
-category: Tires
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
<Lada,Samara,1987>
<Trabant,super,1966>
----------------------

Part: Universal Car Engine
-code: EU33
-category: Engine
-buyPrice: 6733.17
-sellPrice: 6800.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
<Mazda,MX5,1999>
<Mercedes,C500,2008>
<Opel,Astra,1997>
----------------------
```

Testing the Solution

At the end we need to **test our code**. In fact we have done this in the class **`TestShop`**. This doesn't mean that we have tested entirely our problem. We have to **check the border cases**, for example when some of the lists are empty. Let's make a little change of the code in **`Main()`** method, to start the program with an **empty list**:

```
static void Main()
{
    Shop emptyShop = new Shop("Empty Shop");
    Console.WriteLine(emptyShop);
```

```
    Manufacturer lada = new Manufacturer("Lada",
        "Russia", "Moscow", "653443", "893321");
    Part tires = new Part("Tires 165/50/R13", 302.36m,
        345.58m, lada, "T332", PartCategory.Tires);

    Manufacturer bmw = new Manufacturer("BWM",
        "Germany", "Bavaria", "665544", "876666");
    Part engineOil = new Part("BMW Engine Oil",
        633.17m, 670.0m, bmw, "Oil431", PartCategory.Engine);
    engineOil.AddSupportedCar(new Car("BMW", "316i", 1994));

    Shop ultraTuningShop = new Shop("Ultra Tuning Shop");
    ultraTuningShop.AddPart(tires);
    ultraTuningShop.AddPart(engineOil);

    Console.WriteLine(ultraTuningShop);
}
```

The result of this test is:

```
Shop: Empty Shop


Shop: Ultra Tuning Shop

Part: Tires 165/50/R13
-code: T332
-category: Tires
-buyPrice: 302.36
-sellPrice: 345.58
-manufacturer: Lada <Russia,Moscow,653443,893321>
---Supported cars---
----------------------

Part: BMW Engine Oil
-code: Oil431
-category: Engine
-buyPrice: 633.17
-sellPrice: 670.0
-manufacturer: BWM <Germany,Bavaria,665544,876666>
---Supported cars---
<BMW,316i,1994>
----------------------
```

From the result it seems the first shop is empty and in the second shop the list of cars for the first part is empty. This is the **correct output**. Therefore our program works correctly with the border case of empty lists.

We can continue testing with **other border cases** (e.g. missing part name, missing price, missing manufacturer, etc.), as well as with some kind of **performance test** (e.g. shop with 300,000 parts for 5,000 cars and 200 manufacturers). We will leave this for the readers.

Exercises

1. You are given an input file **mails.txt**, which contains names of users and their email addresses. Each line of the file looks like this:

```
<first name> <last name> <username>@<host>.<domain>
```

   There is a requirement for email addresses – **<username>** can be a sequence of Latin letters (**a-z**, **A-Z**) and underscore (**_**), **<host>** is a sequence of lower Latin letters (**a-z**), and **<domain>** has a limit of 2 to 4 lower Latin letters (**a-z**). Following the guidelines for problem solving write a program, **which finds the valid email addresses** and writes them together with the names of the users (in the same format as in the input) to an output file **valid-mails.txt**.

   Sample input file (**mails.txt**):

```
Steve Smith steven_smith@yahoo.com
Peter Miller pm<5.gmail.com
Svetlana Green svetlana_green@hotmail.com
Mike Johnson mike*j@888.com
Larry Cutts larry.cutts@gmail.com
Angela Hurd angel&7@freemail.hut.fi
```

   Output file (**valid-mails.txt**):

```
Steve Smith steven_smith@yahoo.com
Svetlana Green svetlana_green@hotmail.com
Larry Cutts larry.cutts@gmail.com
```

2. You are given a **labyrinth**, which consists of **N x N squares**, and each of them can be passable (**0**) or not (**x**).

   In one of the squares our hero Jack (*) is positioned. Two squares are **neighbors**, if they have a common wall. At one step Jack can pass from one passable square to its neighboring passable square. Write a program, which prints **the number of possible exits** from given labyrinth. At the figure below we have **7 possible exits**, reachable from the start position.

x	x	x	0	x	x
0	x	0	0	0	
0	*	0	x	0	0
x	x	x	x	0	x
0	0	0	0	0	x
0	x	0	x	x	0

The **input data** is read from a text file named `Labyrinth.in`. At the first line in the file is the number **N** (2 < N < 1000). At the next N lines there are N characters, each either "**0**" or "**x**" or "*****". The output is a single number and should be printed in the file `Labyrinth.out`.

3. You are given a **labyrinth**, which consists of **N x N squares**, each of it can be passable or not. Passable cells consist of lower Latin letter between "**a**" and "**z**", and the non-passable – '**#**'. In one of the squares is Jack. It is marked with "*****".

   Two squares are **neighbors**, if they have common wall. At one step Jack can pass from one passable square to its neighboring passable square. When Jack passes through passable squares, he writes down the letters from each square. At each exit he **gets a word**. Write a program, which from a given labyrinth prints the words, which Jack gets from all the possible exits. At the example below Jack can get **10 different words** corresponding to its 10 possible paths he could find to some of the exits: *a*, *az*, *aza*, *madk*, *madkm*, *madam*, *madamk*, *dir*, *did*, *difid*.

a	#	#	k	m	#
z	#	a	d	a	#
a	*	m	#	#	#
#	d	#	#	#	#
r	i	f	i	d	#
#	d	#	d	#	t

   The input data is read from a text file named `Labyrinth.in`. At the first line in the file there is the number N (2 < N < 10). At each of the next N lines there are N characters, each of them is either Latin letter between "**a**" and "**z**" or "**#**" (impassable wall) or "*****" (Jack). The output must be printed in the file `Labyrinth.out`.

4. A company plans to create a **system for managing of a sound recording company**. The sound recording company has a name, address, owner and performers. Each **performer** has name, nickname and created albums. **Albums** are described with name, genre, year of creation, number of sold copies and list of songs. The **songs** are described with name and duration. Design a **set of classes with**

relationships between each other, which models the data of the record company. Implement a test class, which demonstrates the work of rest of the classes.

5. A company plans on **creating of a system for managing a company for real estates**. The company **has** name, owner, tax ID, employees and has a list of estates for sale. **Employees** are described with name, work position and experience. The company sells several **types of estates**: apartments, houses, undeveloped areas and shops. All of them are characterized with area, price of square meters and location. For some of them there is additional information. For the **apartments** there is data about the number of the floor, whether there is an elevator in the block, and if it is furnished. For the **houses** the data is – square meters for the undeveloped area and for the developed (yard), how many floors it has and whether it is furnished. **Design a set of classes with relationships** between them, which model the data for the company. **Implement a test class**, which demonstrates the work of the rest of the classes.

Solutions and Guidelines

1. The problem is similar to the first problem from our sample exam. Again we can read line by line the input file and with appropriate **regular expression** to check the email addresses. **Test the solution** carefully before you go to the next problem.

2. **Possible exits** from the labyrinth are all the cells, which are positioned at the border of the labyrinth and are reachable from the initial cell. The problem could be solved using **BFS** with **just little modification** of the solution of the **"Escape from Labyrinth"**. **Test your solution** carefully!

3. The problem is similar to the previous one, but **all possible paths** to the exit are required. You can do recursive search with **backtracking** (**DFS**) and keep in a **StringBuilder** the letters to the exit, to create the words, which you have to print. With bigger labyrinths the problem has no optimal solution (there is no way to print all the paths, without generating all of them, but they **grow exponentially** to the labyrinth size). **Test** carefully your solution and think of special cases that need special care.

4. You must write the required classes: **MusicCompany**, **Artist**, **Album**, **Song**. Think of the links between classes and what data structures to use for them. For the printing redefine the method **ToString()** from **System.Object**. **Test all methods** and the border cases.

5. The classes you must write are **EstateCompany**, **Employee**, **Apartment**, **House**, **Shop** and **UndevelopedArea**. Export all shared characteristics in separate **abstract base class Estate**. Encapsulate all fields with properties. Override the method **ToString()**, which to collect the data of the corresponding class and print it to the console. **Test all methods** and special border cases (like missing property values).

Chapter 25. Sample Programming Exam – Topic #2

In This Chapter

In this chapter, we will take a look at the specifications of a **few practical algorithmic problems** from a sample programming exam, and we will offer solutions. While solving the problems, we will follow the guidelines from the chapter "Problem Solving Methodology", and we are going to illustrate their implementation.

Problem 1: Counting the Uppercase / Lowercase Words in a Text

Write a program that **counts the words in a text** entered from the console. The program must output the **total number of words**, the number of words written in **uppercase** and the number of words written in **lowercase**. If a word appears more than once in the text, each repetition counts as a new occurrence. Every character that is not a letter counts as a word separator.

Sample input:

```
Welcome to your first programming exam! Can you think of a
solution to this problem and write it down? GOOD LUCK!
```

Sample output:

```
Word count: 21
Upper case words: 2
Lower case words: 17
```

Coming Up with an Appropriate Idea for a Solution

Intuitively, it comes to mind, that the problem may be solved by **splitting the text up into separate words and counting** those that meet the specified conditions.

Obviously, this approach is correct, but far too general, and it doesn't lead to a particular method for solving the problem. Let's try to **be more specific**, and see if by doing so, we could implement an algorithm that will lead to a

solution. It might turn out that the implementation is difficult, or that the complexity of the solution is too great for the program to complete its execution even with today's powerful computers. If that is the case, we would have to find another solution to the problem.

Breaking Down the Problem into Subproblems

A useful approach for solving algorithmic problems is to try **breaking them down into smaller problems** that are easier and quicker to solve. Let's try defining the necessary steps for solving this problem.

First of all, we have to **split the text up into separate words**. This, in and of itself, is not a simple task, but it is the first step towards breaking down the problem into smaller, although still complicated, subproblems.

Then we need to **count the words** that concern us. This is the second major problem we have to solve. Let's take a look at both problems separately and try breaking them down even further.

How Do We Split the Text Up into Separate Words?

In order to split the text up into separate words, we need to find a way to identify them first. According to the problem specifications every non-letter character functions as a word separator. Therefore, we must first **identify these separators** and use them to split the text in tokens.

So far, we have formulated two subproblems – **finding the separators** and **partitioning (splitting) the text** in accordance with the characters found. We can implement their solutions right away. This was in fact our goal from the start – breaking down complicated problems into smaller and easier subproblems.

In order to **find the separators**, all we need to do is iterate through all characters and extract those that aren't letters.

Once we have identified the separators, we can implement the text partitioning by invoking the **`Split(…)`** method of the **`String`** class.

How Do We Count the Words?

Let's assume we already have a list of all words from the text. We want to **find the total word count, the number of words in uppercase and the number of words in lowercase**.

To do this, we can go through each and every word from the list and check if it meets either of the necessary conditions. At each step we **increment the total word count**. We check if the current word is in uppercase and, if so, we **increment the number of words in uppercase**. Likewise, we check if the word consists only of lowercase letters and **increment the lowercase word counter**.

Thus, we have defined another **two subproblems** – recognizing uppercase and lowercase words. These appear to be very easy. It might even turn out

that the **string** class provides such functionality. After we check, it turns out this is not the case. Yet we notice that there are methods that allow us to convert a string to an uppercase or a lowercase string. This might be of use.

To check if a word consists only of uppercase letters, all we have to do is compare it to the string resulting after converting the word to uppercase. If the two are equal, then the comparison returns true. Performing the check for lowercase words is done likewise.

Verifying the Idea

It seems our idea is a good one. We've bro**ken down the problem into subproblems** and we know how to solve each of them. Should we continue towards the implementation? Haven't we overlooked something?

Shouldn't we have verified the idea by writing down a few examples on paper? Perhaps we would come across something we have missed. We could start with the example given in the problem statement:

```
Welcome to your first programming exam! Can you think of a
solution to this problem and write it down? GOOD LUCK!
```

The separators would be: **spaces**, **?** and **!**. The words that have come up are the following: **Welcome**, **to**, **your**, **first**, **programming**, **exam**, **Can**, **you**, **think**, **of**, **a**, **solution**, **to**, **this**, **problem**, **and**, **write**, **it**, **down**, **GOOD**, **LUCK**.

Counting the words we acquire the correct result. It seems the idea is adequate, and **it works**. Now we can proceed towards implementing it. We will do this step by step and at each step we will implement one subproblem.

Let's Consider the Data Structures

The problem is simple and doesn't need complex data structures.

We can use the **char** data type for storing each separator. During the process of finding the separator characters we add each of them to a list. We can use either **char[]** or **List<char>**. In this case, we will choose the latter.

As for the words in the text, we can use an array of strings **string[]** or **List<string>**.

Let's Consider the Efficiency

Are there any **performance requirements**? How long can the text be?

Since the text will be entered from the console, it's unlikely to be very long. No one is going to type 1MB of text into the console. We can assume that the solution's **performance is not critical**.

Let's Write Down the Solution

It's very good practice to **write the solution down on a piece of paper** before typing it on the computer. This helps uncover drawbacks in our idea or implementation beforehand. In addition, implementing the solution will be considerably quicker, because of the outlines we can provide and because we would then have a better grasp of both the problem and the solution.

Step 1: Finding the Separators in the Text

We will define a method that **extracts all non-letter characters** from the text and return them as an array of characters. Then we will use that array for splitting the text up into separate words. We will use **`List<char>`** to keep the separators we find when passing through the text:

```
private static char[] ExtractSeparators(string text)
{
   List<char> separators = new List<char>();
   foreach (char character in text)
   {
      // If the character is not a letter,
      // then by definition it is a separator
      if (!char.IsLetter(character))
      {
         separators.Add(character);
      }
   }
   return separators.ToArray();
}
```

We use a loop to iterate through all of the characters in the text. We check if the current character is a letter by invoking the **`IsLetter()`** method of the primitive data type **`char`**. If it's not, we add the character to the separators. Finally, our method returns an array containing the separators.

Testing the ExtractSeparators(…) Method

Before we go any further, it's advisable to **test if extracting the separators is working correctly**. For this purpose, we will write two additional methods. The first of these is **`TestExtractSeparators()`** which will test the execution of **`ExtractSeparators(…)`** and the second – **`GetTestData()`** – will return different texts, which will allow us to test our solution:

```
private static void TestExtractSeparators()
{
   List<string> testData = GetTestData();
   foreach (string testCase in testData)
```

```
  {
    Console.WriteLine(
      "Test Case:{0}{1}", Environment.NewLine, testCase);
    Console.WriteLine("Result:");
    foreach (char separator in ExtractSeparators(testCase))
    {
      Console.Write("{0} ", separator);
    }
    Console.WriteLine();
  }
}

private static List<string> GetTestData()
{
  List<string> testData = new List<string>();
  testData.Add("This is wonderful!!! All separators like " +
    "these ,.(? and these /* are recognized. It works.");
  testData.Add("SingleWord");
  testData.Add(string.Empty);
  testData.Add(">?!>?#@?");
  return testData;
}

static void Main()
{
  TestExtractSeparators();
}
```

We start the program and **check if the separators have been correctly identified**. The first test's result is as follows:

```
Test Case:
This is wonderful!!! All separators like these ,.(? and these /*
are recognized.
 It works.
Result:
    ! ! !           , . ( ?       / *     .     .
Test Case:
SingleWord
Result:

Test Case:

Result:
```

```
Test Case:
>?!>?#@?
Result:
> ? ! > ? # @ ?
```

We might think of the above output as **partially correct**. In fact it does extract correctly the separators between the words but most of them are **duplicated several times**. We need all the separators without duplications, right?

Correcting the ExtractSeparators(…) Method

To correct the method for extracting the separators between the words in the text, we can use a different data structure to keep them. We know that **sets** keep elements without duplications. So we could use **`HashSet<char>`** instead of **`List<char>`** to hold the separator characters we find in the text:

```
private static char[] ExtractSeparators(string text)
{
  HashSet<char> separators = new HashSet<char>();
  foreach (char character in text)
  {
    // If the character is not a letter,
    // then by definition it is a separator
    if (!char.IsLetter(character))
    {
      separators.Add(character);
    }
  }
  return separators.ToArray();
}
```

The code is almost the same, but we use a set instead of list to avoid duplicated separators. We might need to include the **`System.Linq`** namespace in the start of the program to use the **`ToArray()`** extension method for converting a hash set to an array.

Testing Again after the Fix

We test the above method with the same testing code and we find **it now works correctly**. The separators are extracted correctly with no duplicates:

```
Test Case:
This is wonderful!!! All separators like these ,.(? and these
/* are recognized.
 It works.
```

```
Result:
  ! , . ( ? / *
Test Case:
SingleWord
Result:

Test Case:

Result:

Test Case:
>?!>?#@?
Result:
> ? ! # @
```

We test also with some **borderline cases** – text consisting of a single word without separators; text consisting of separators only; an empty string. We've already included such tests in our **GetTestData()** method. It seems that the method works fine and we can proceed to the next step.

Step 2: Splitting Up the Text in Separate Words

We will use **string**'s **Split(…)** method with the specified separators for splitting up the text by the separators and extracting the words from it. This is how our method looks like:

```
private static string[] ExtractWords(string text)
{
  char[] separators = ExtractSeparators(text);
  string[] words = text.Split(separators,
    StringSplitOptions.RemoveEmptyEntries);
  return words;
}
```

Testing the Word Extracting Method

Before we carry on to the next step, we have to see if the method works correctly. To do this, we will reuse the **GetTestData()** for the input test data and we will test the new **ExtractWords(…)** method:

```
private static void TestExtractWords()
{
  List<string> testData = GetTestData();
  foreach (string testCase in testData)
  {
```

```
        Console.WriteLine("\nTest Case: {0}", testCase);
        string[] words = ExtractWords(testCase);
        Console.WriteLine("Result: {0}", string.Join(" ", words));
    }
}

static void Main()
{
    TestExtractWords();
}
```

The result from the above test looks **correct**:

```
Test Case: This is wonderful!!! All separators like these ,.(?
and these /* are
recognized. It works.
Result: This is wonderful All separators like these and these
are recognized It
works

Test Case: SingleWord
Result: SingleWord

Test Case:
Result:

Test Case: >?!>?#@?
Result:
```

We check the results from the other test cases. We verify that they are **correct** and that our algorithm is accurate (till this stop).

Step 3: Determining Whether a Word Is in Uppercase or Lowercase

We already have an idea how to implement the **uppercase / lowercase checks**, and we can write the corresponding methods directly:

```
private static bool IsUpperCase(string word)
{
    bool result = word.Equals(word.ToUpper());
    return result;
}

private static bool IsLowerCase(string word)
```

```
{
    bool result = word.Equals(word.ToLower());
    return result;
}
```

We **test the above methods** by passing words in uppercase, lowercase and mixed case. The results are **correct**.

Step 4: Counting the Words

Now we can proceed to **solving the problem itself** – counting the words. All we have to do is iterate through the list of words and depending on the word's type to increment the corresponding counters. Then we print the result:

```
private static void CountWords(string[] words)
{
    int allUpperCaseWordsCount = 0;
    int allLowerCaseWordsCount = 0;
    foreach (string word in words)
    {
        if (IsUpperCase(word))
        {
            allUpperCaseWordsCount++;
        }
        else if (IsLowerCase(word))
        {
            allLowerCaseWordsCount++;
        }
    }

    Console.WriteLine("Total words count: {0}", words.Length);
    Console.WriteLine("Upper case words count: {0}",
        allUpperCaseWordsCount);
    Console.WriteLine("Lower case words count: {0}",
        allLowerCaseWordsCount);
}
```

Testing the Word Counting Method

Let's **check if we count the words correctly**. We will write another test method using the data from the **GetTestData()** method and the previously written and tested **ExtractWords(…)** method:

```
private static void TestCountWords()
{
```

```
    List<string> testData = GetTestData();
    foreach (string testCase in testData)
    {
        Console.WriteLine("Test Case: {0}", testCase);
        Console.WriteLine("Result: ");
        CountWords(ExtractWords(testCase));
        Console.WriteLine();
    }
}

static void Main()
{
    TestCountWords();
}
```

Executing the application, we obtain the **correct result**:

```
Test Case: This is wonderful!!! All separators like these ,.(? and these /* are recognized. It works.
Result:
Total words count: 13
Upper case words count: 0
Lower case words count: 10

Test Case: SingleWord
Result:
Total words count: 1
Upper case words count: 0
Lower case words count: 0

Test Case:
Result:
Total words count: 0
Upper case words count: 0
Lower case words count: 0

Test Case: >?!>?#@?
Result:
Total words count: 0
Upper case words count: 0
Lower case words count: 0
```

The above **results are correct** (the **typical case** and a few **borderline cases**). We perform few other **borderline tests**, e.g. when the list contains words in uppercase or lowercase only, or when the list is empty. All of them work correctly.

Note that it is a good idea to use **unit testing** instead of these semi-automated tests. Recall how we **write unit tests in Visual Studio** (in the chapter "High-Quality Code") and try to convert our test methods to unit tests for the Visual Studio Team Test (VSTT) framework.

Step 5: Console Input

All that's left to implement is **the final step** – allowing the user to input text:

```
private static string ReadText()
{
  Console.WriteLine("Enter text:");
  return Console.ReadLine();
}
```

Note that as a rule unless the input comes from a text file or is very short (e.g. just one number or few characters) **it should be read as a final step**. Otherwise we will need to enter the input data each time when we start the program and this will waste a lot of time and can lead to errors.

Step 6: Putting All Together

Now after **all subproblems have been solved**, we can proceed to the complete solution to the problem. We need to add a **Main(…)** method, which will **combine together the different parts of the solution**:

```
static void Main()
{
  string text = ReadText();
  string[] words = ExtractWords(text);
  CountWords(words);
}
```

Testing the Solution

While implementing the solution, we **wrote test methods for every method**, integrating them with each other gradually. For the moment, we are certain they interact correctly; there's nothing we have overlooked and there is no method that does unnecessary work or that returns incorrect results.

If we would like to **test the solution with more data**, we would only need to add it to the **GetTestData(…)** method. If we want, we may even rewrite the **GetTestData(…)** method so that it reads the test data from an external source, e.g. from a text file.

Here's how the final solution looks like at the end:

WordsCounter.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

public class WordsCounter
{
  static void Main()
  {
    string text = ReadText();
    string[] words = ExtractWords(text);
    CountWords(words);
  }

  private static string ReadText()
  {
    Console.WriteLine("Enter text:");
    return Console.ReadLine();
  }

  private static char[] ExtractSeparators(string text)
  {
    HashSet<char> separators = new HashSet<char>();
    foreach (char character in text)
    {
      // If the character is not a letter,
      // then by definition it is a separator
      if (!char.IsLetter(character))
      {
        separators.Add(character);
      }
    }
    return separators.ToArray();
  }

  private static string[] ExtractWords(string text)
  {
    char[] separators = ExtractSeparators(text);
    string[] words = text.Split(separators,
      StringSplitOptions.RemoveEmptyEntries);
    return words;
  }

  private static bool IsUpperCase(string word)
  {
```

```
        bool result = word.Equals(word.ToUpper());
        return result;
    }

    private static bool IsLowerCase(string word)
    {
        bool result = word.Equals(word.ToLower());
        return result;
    }

    private static void CountWords(string[] words)
    {
        int allUpperCaseWordsCount = 0;
        int allLowerCaseWordsCount = 0;

        foreach (string word in words)
        {
            if (IsUpperCase(word))
            {
                allUpperCaseWordsCount++;
            }
            else if (IsLowerCase(word))
            {
                allLowerCaseWordsCount++;
            }
        }

        Console.WriteLine("Total words count: {0}", words.Length);
        Console.WriteLine("Upper case words count: {0}",
            allUpperCaseWordsCount);
        Console.WriteLine("Lower case words count: {0}",
            allLowerCaseWordsCount);
    }
}
```

We removed the testing methods from our code to simplify it. The best practice is instead of removing the tests to **create a separate testing project** and put all the tests in a **testing class**. This is best achieved though the Visual Studio's unit testing framework, as it was shown in the chapter "High-Quality Code".

A Word on Performance

Since there are **no explicit performance requirements**, we will only make a suggestion for dealing with the situation when the algorithm turns out to be slow. Splitting the text with separators assumes that the **entire text will be**

loaded into memory. The list of words, after partitioning the text, will also be written to memory. Therefore, if the input text is large, the program will also **consume a large amount of memory**. For example, if the input text is 200MB long, then the program will consume at least 800MB of memory, because each word is stored as 2 bytes for every character (.NET uses UTF-16 character encoding for the strings in memory).

If we want to **avoid high memory consumption** then the words must not be stored in memory all at once. We can come up with **another algorithm**: **scanning the text char by char** and storing the letters into a buffer (such as `StringBuilder`). If at a certain moment a separator is encountered, then the buffer contains the most recent word. We can analyze its casing and then empty the buffer. We can repeat this until the end of the file is reached. This appears to be **more efficient**, doesn't it?

A **more efficient lower / upper case checker** would be to iterate through all letters using a loop and to examine them char by char. That way we can skip a lower / upper case conversion, which allocates extra memory for every word. After the word has been processed, the memory will be freed, which would eventually lead to extra CPU utilization (for the .NET garbage collector).

Obviously, the latter solution is **more efficient**. The question is if we should scrap the original solution and write a completely different one. It all **depends on the performance requirements**. The problem description doesn't hint at an input text measuring in the hundreds of megabytes. Therefore the current solution, although not optimal, is still correct and will suffice. We suggest the **reader to implement the proposed fast solution** and to **compare how faster it is**, e.g. by processing an input of 100 MB.

Problem 2: A Matrix of Prime Numbers

Write a program that reads a positive integer N from the standard input and **prints the first N^2 prime numbers as a square matrix of size N x N**. The matrix must be filled with numbers starting from the first row and ending at the last one. Each row must be filled with prime numbers from left to right.

Note: A **prime number** is a number that has no divisors other than 1 and itself. The number 1 is not a prime number.

Sample input:

2	3	4

Sample output:

2 3 5 7	2 3 5 7 11 13 17 19 23	2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53

Coming Up with an Appropriate Idea for a Solution

We can solve the problem by printing the rows and columns of the resulting matrix using **two nested loops**. For each of its elements we will extract and print the corresponding **prime number**.

Breaking Down the Problem into Subproblems

We must solve at least **two subproblems** – **finding each successive prime number** and **printing the prime numbers into a matrix**. We can print the matrix right away, but the process of finding each successive prime number will require additional thinking. Perhaps the most intuitive way to accomplish this is to start testing the primality of each number starting from the last prime number that we found. When a new prime is encountered, it is returned as a result. Thus, a new subproblem has come up – checking whether or not a number is a prime.

Verifying the Idea

Our idea for a solution leads directly to the required result. We **write down a couple of examples on a piece of paper** and make sure that it works.

Consider the Data Structures

The problem makes use of one **data structure** only – a **matrix**. It's only natural to use a two-dimensional array (matrix).

Consider the Efficiency

Displaying at the console large matrices (for example of size 1000 x 1000) cannot be properly handled. This means our solution should work for reasonably large matrices, e.g. on the order of N ≤ 200. We don't need to consider cases where the matrix is too large. When N = 200, our algorithm will find **the first 40,000 prime numbers** and should not run slowly.

Now we are ready for the **implementation of the algorithm** we invented.

Step 1: Check to Find If a Number Is a Prime

To test a number for primality, we can define a method called **IsPrime(…)**. The test will verify that dividing the number by any of its predecessors always yields a division remainder. To be more precise, it is sufficient to **check the integers between 2 and the square root of the number**. This holds true, because if the number **p** has a divisor **x**, then **p = x.y**, and at least one or both of the numbers **x** and **y** will be less than or equal to the square root of **p**. What follows is an implementation of the method:

```
private static bool IsPrime(int number)
{
```

```
    int maxDivider = (int)Math.Sqrt(number);
    for (int divider = 2; divider <= maxDivider; divider++)
    {
        if (number % divider == 0)
        {
            return false;
        }
    }
    return true;
}
```

The **algorithm complexity** of the above example is **O(sqrt(number))**, because the amount of checks that will be made is not greater than the square root of the number. This complexity will suffice for the problem at hand, but is it possible to optimize this method even further?

Come to think about it, every second number is even and all even numbers are divisible by 2. In that case, if the number we are testing is odd, the above method will needlessly check all odd numbers from 2 to the square root of the number. How can we omit these unnecessary checks? We could find out if the number is even at the very beginning of the method. If it's not, it will be processed in a modified version of the main loop that skips even numbers. Using this new approach, we have achieved the same computational complexity of **O(sqrt(number))**, but with a better constant 1/2.

This example illustrates how to optimize a bit the existing method:

```
private static bool IsPrime(int number)
{
    if (number == 2)
    {
        return true;
    }
    if (number % 2 == 0)
    {
        return false;
    }

    int maxDivider = (int)Math.Sqrt(number);
    for (int divider = 3; divider <= maxDivider; divider += 2)
    {
        if (number % divider == 0)
        {
            return false;
        }
    }
```

```
    return true;
}
```

As we can see, there was a minimal amount of changes compared to the non-optimized version.

Testing the Prime Checking Method

We can make sure both methods work correctly by consecutively passing to them different numbers, some of which will be primes, and verifying the results.

	Always test a method to make sure it works before optimizing it.

The reason to test your methods is that after optimization, the code usually gets longer, more difficult to read and therefore more difficult to debug, if it is incorrect.

	Be careful when optimizing a piece of code. Do not go to extremes by making unnecessary optimizations that will make your program marginally faster at the expense of readability and maintenance.

To check the prime checking method we could write a piece of code like this:

```
static void Main()
{
    Console.WriteLine(IsPrime(2));
    Console.WriteLine(IsPrime(3));
    Console.WriteLine(IsPrime(4));
    Console.WriteLine(IsPrime(5));
    Console.WriteLine(IsPrime(121));
}
```

It runs as expected and the produced results are correct:

```
True
True
False
True
False
```

Step 2: Finding the Next Prime Number

In order to find the **next prime number**, we can define a method that takes an integer as a parameter and returns the first prime number equal or larger

than it. To check if the number is prime, we will use the method from the previous step. Below is an implementation of the method:

```
private static int FindNextPrime(int startNumber)
{
   int number = startNumber;
   while (!IsPrime(number))
   {
      number++;
   }
   return number;
}
```

Testing the Next Prime Number Finder

Once again we have to test the method by passing a few numbers and verifying that the result is correct:

```
static void Main()
{
   Console.WriteLine(FindNextPrime(2));
   Console.WriteLine(FindNextPrime(3));
   Console.WriteLine(FindNextPrime(4));
   Console.WriteLine(FindNextPrime(5));
   Console.WriteLine(FindNextPrime(121));
}
```

The result is correct, as expected:

```
2
3
5
5
127
```

Step 3: Printing the Matrix

Now that we have defined the previous methods, we are all set to **print the entire matrix of prime numbers:**

```
private static void PrintMatrixOfPrimes(int dimension)
{
   int lastPrime = 1;
   for (int row = 0; row < dimension; row++)
   {
      for (int col = 0; col < dimension; col++)
```

```
    {
      int nextPrime = FindNextPrime(lastPrime + 1);
      Console.Write("{0,4}", nextPrime);
      lastPrime = nextPrime;
    }
    Console.WriteLine();
  }
}
```

We will test this method as part of testing the entire program.

Step 4: Console Input

All that's left is to add functionality allowing us to **read N from the console**.

```
static void Main()
{
  int n = ReadInput();
  PrintMatrixOfPrimes(n);
}

private static int ReadInput()
{
  Console.Write("N = ");
  string input = Console.ReadLine();
  int n = int.Parse(input);
  return n;
}
```

Testing the Entire Solution

After we have completed all other steps, we can proceed with **testing the entire solution**. To do this, we could look up the first 25 prime numbers (at a sheet of paper) and test the program's output for values of N between 1 and 5. We should include special **border cases** like N=0 and N=1. We know that at border cases the likelihood of making a mistake is significantly higher.

In our case, we can confine ourselves with the examples from the problem description, provided that the methods have been thoroughly tested at each step. This is the output of the program for N = 1, 2, 3 and 4 respectively:

```
2         2 3              2 3 5                 2 3 5 7
          5 7              7 11 13               11 13 17 19
                           17 19 23              23 29 31 37
                                                 41 43 47 53
```

The result is correct and after few more tests we get convinced that we have **solved correctly the problem** "Matrix of Prime Numbers".

We can make sure the solution works relatively fast even for larger values of N. For example, there is no perceived lag for N = 200.

A Word on Performance

We should point out, that our solution **does not find prime numbers in the most efficient way**. Despite of this drawback, due to the solution's clarity and the reasonably small size of the matrix, we can utilize this algorithm without performance issues.

Improved Performance: Sieve of Eratosthenes

If we have to **improve the performance**, we can find the first N^2 prime numbers using the Sieve of Eratosthenes. That way we will not need to check if every number is prime until we find N^2 prime numbers. You might ask yourself **how large Eratosthenes's Sieve** will be needed if we want to find the first N^2 prime numbers. You might use the following approximation (without any mathematical proof):

```
long sieveSize =
   (long)Math.Truncate(2.4 * n * n * Math.Log(n, Math.E)) + 2;
```

If the Eratosthenes's Sieve is at least **sieveSize** elements large, it will be enough to produce the first N^2 prime numbers and not too much above them. You could check this manually or **you might invite a better formula** using complex mathematical calculations (see http://en.wikipedia.org/wiki/Prime-counting_function). For example if N=10, the estimated **sieveSize** will be 554 and it will find the first 101 prime numbers (we need 10^2 = 100 prime numbers to fill the matrix, so these 101 prime numbers are enough). If N=1,000, the **sieveSize** will be 16,578,614 and it will find the first 1,065,855 prime numbers. For N=5,000 the **sieveSize** will be 511,031,593 and it will find the first 26,905,486 prime numbers. For significantly bigger sizes Eratosthenes's Sieve will not fit in the memory. You might **try to implement this algorithm** and **check how faster it is**. When comparing the speed, you may redirect the output to a file to save some time which is spent in printing the matrix.

Problem 3: Evaluate an Arithmetic Expression

Write a program that **evaluates a simple arithmetic expression** consisting of unsigned integers and the arithmetic operations "**+**" and "**-**". There will be no blank spaces between the integers.

The expression will have the following format:

```
<number><operation>…<number>
```

Sample input:

```
1+2-7+2-1+28+2+3-37+22
```

Sample output:

```
15
```

Coming Up with an Appropriate Idea for a Solution

To solve this problem, we can take advantage of the strict expression format, which guarantees we have a sequence of a number, operation, another number and so on.

That way, we can first **extract all numbers** from the expression, and then we can **extract all operators** and finally evaluate the result by **combining the numbers with the operators**.

Verifying the Idea

Sure enough, if we test this approach with a few expressions **using pen and paper**, we acquire a correct result. Initially, the result is equal to the first number and at each step we either add or subtract the next number depending on the current operator.

Data Structures and Efficiency

The problem is too simple for us to use complex **data structures**. The numbers and characters can be stored in **arrays**. Performance issues are out of the question, because the characters and numbers are processed exactly once, i.e. the complexity of the algorithm is **linear**. Even with millions of integers and operators, the algorithm is expected to work fast.

Breaking Down the Problem into Subproblems

Now that we have made sure the idea works, we can move on to breaking down the problem into subproblems. The first subproblem we will have to solve is **extracting the numbers** from the expression. The second – **extracting the operators**. Finally, we will **evaluate the entire expression** using the extracted numbers and operators.

Step 1: Extracting the Numbers

In order to extract the numbers, we need to **split the expression using the operators (+ and -) as separators**. This is easily done using the **`Split(…)`** method of the **`string`** class. Afterwards, we have to convert the resulting array of strings to an array of integers:

```
private static int[] ExtractNumbers(string expression)
{
  string[] splitResult = expression.Split('+', '-');
  int[] resultNumbers = new int[splitResult.Length];
  for (int i = 0; i < splitResult.Length; i++)
  {
    resultNumbers[i] = int.Parse(splitResult[i]);
  }
  return resultNumbers;
}
```

We use the **Parse(…)** method of the **Int32** class to convert strings to integers. It takes a string as a parameter and returns the integer value that the string represents.

Why do we **use an array to store the numbers**? Can't we use a linked list or a dynamic array? Of course we can, but in our case we only need to store the integers and iterate through them when evaluating the result. Therefore, an array is sufficient and is the simplest collection that will work.

Testing the Extraction of Numbers

Before we move on to the next step we should **check if the numbers are extracted correctly**. We may use the following example:

```
static void Main()
{
  int[] numbers = ExtractNumbers("1+2-7+2-1+28");
  foreach (int x in numbers)
  {
    Console.Write("{0} ", x);
  }
}
```

The result is **exactly as it should be**:

```
1 2 7 2 1 28
```

We examine the border case when the expression consists only of one number and no operators, and we make sure it is handled properly.

Step 2: Extracting the Operators

We can **extract the operators** by iterating through each consecutive character from the string and check if it is one of the specified operators:

```
private static char[] ExtractOperators(string expression)
```

```
{
   string operatorCharacters = "+-";
   List<char> operators = new List<char>();
   foreach (char c in expression)
   {
      if(operatorCharacters.Contains(c))
      {
         operators.Add(c);
      }
   }
   return operators.ToArray();
}
```

Testing the Extraction of Operators

Here's how we **test** whether or not the method works correctly:

```
static void Main()
{
   char[] operators = ExtractOperators("1+2-7+2-1+28+3+1");
   foreach (char oper in operators)
   {
      Console.Write("{0} ", oper);
   }
}
```

The output after the program's execution is **correct**:

```
+ - + - + + +
```

We create a **test for the border case** when the expression consists of only one number and no operators. Just as expected, we get an empty array and the output of the above testing program is and empty string.

Step 3: Evaluating the Expression

When **evaluating the expression**, we can make use of the fact that the **numbers' count is always greater than the operators' count by one**. Using a single loop we can evaluate the expression, provided we have the lists of numbers and operators:

```
private static int CalculateExpression(int[] numbers,
   char[] operators)
{
   int result = numbers[0];
   for (int i = 1; i < numbers.Length; i++)
```

```
    {
        char operation = operators[i - 1];
        int nextNumber = numbers[i];
        if (operation == '+')
        {
            result += nextNumber;
        }
        else if (operation == '-')
        {
            result -= nextNumber;
        }
    }
    return result;
}
```

Test the Evaluation of Expression

We test the method's execution:

```
static void Main()
{
    // Expression: 1 + 2 - 3 + 4
    int[] numbers = new int[] { 1, 2, 3, 4 };
    char[] operators = new char[] { '+', '-', '+' };
    int result = CalculateExpression(numbers, operators);
    // Expected result is 4
    Console.WriteLine(result);
}
```

The result seems to be **correct**:

```
4
```

We perform few other tests (e.g. a 1, 1+2, 1-1) and it still works correctly.

Step 4: Console Input

We have to provide the user with the means to **enter an expression**:

```
private static string ReadExpression()
{
    Console.Write("Enter expression: ");
    string expression = Console.ReadLine();
    return expression;
}
```

This could be **left untested**, because it is too simple and it will be tested later when the entire program is finished and gets tested.

Step 5: Putting All Together

All that's left is to **integrate all methods** so that they can function together.

```
static void Main()
{
   string expression = ReadExpression();

   int[] numbers = ExtractNumbers(expression);
   char[] operators = ExtractOperators(expression);

   int result = CalculateExpression(numbers, operators);
   Console.WriteLine("{0} = {1}", expression, result);
}
```

Testing the Solution

We can use the **example from the problem description** when testing the solution. We get a correct result:

```
Enter expression: 1+2-7+2-1+28+2+3-37+22
1+2-7+2-1+28+2+3-37+22 = 15
```

In order to make sure the solution runs correctly, we must create a **few more tests** with different parameters, which will include the case when the expression consists of **one number only**.

We can even test with an **empty string**. It's not clear if such input is valid, but we can take it into account just to be on the safe side. It's also ambiguous what would happen, if "**2 + 3**" (with spaces) is entered, instead of "**2+3**". It's a good idea to cope with these situations.

One test case we neglected occurs when a number doesn't fit in an **int** variable. What do you think would happen, if the user enters the expression "**11111111111111111111111111111+222222222222222222222222222222**"?

Small Corrections and Repeated Testing

The program will raise an exception (of type **System.FormatException** or **System.OverflowException** in all likelihood) when the expression is invalid. It's sufficient to catch the exceptions, and when they are thrown, to report that an invalid expression has been entered. Below is the solution's full implementation after this correction:

SimpleExpressionEvaluator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

public class SimpleExpressionEvaluator
{
  private static int[] ExtractNumbers(string expression)
  {
    string[] splitResult = expression.Split('+', '-');
    int[] resultNumbers = new int[splitResult.Length];
    for (int i = 0; i < splitResult.Length; i++)
    {
      resultNumbers[i] = int.Parse(splitResult[i]);
    }
    return resultNumbers;
  }

  private static char[] ExtractOperators(string expression)
  {
    string operationsCharacters = "+-";
    List<char> operators = new List<char>();
    foreach (char c in expression)
    {
      if (operationsCharacters.Contains(c))
      {
        operators.Add(c);
      }
    }
    return operators.ToArray();
  }

  private static int CalculateExpression(
    int[] numbers, char[] operators)
  {
    int result = numbers[0];
    for (int i = 1; i < numbers.Length; i++)
    {
      char operation = operators[i - 1];
      int nextNumber = numbers[i];
      if (operation == '+')
      {
        result += nextNumber;
```

```
            }
            else if (operation == '-')
            {
                result -= nextNumber;
            }
        }
        return result;
    }

    private static string ReadExpression()
    {
        Console.WriteLine("Enter expression:");
        string expression = Console.ReadLine();
        return expression;
    }

    static void Main()
    {
        try
        {
            string expression = ReadExpression();

            int[] numbers = ExtractNumbers(expression);
            char[] operators = ExtractOperators(expression);

            int result = CalculateExpression(numbers, operators);
            Console.WriteLine("{0} = {1}", expression, result);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Invalid expression!");
        }
    }
}
```

To ensure everything works correctly after the fix we need to **test the above code again** with: single number, two numbers, typical expression (like the sample input from the problem description), expression with spaces (e.g. "1 + 2 -3"), expression with large numbers, invalid expression (e.g. -1).

Performance Test

Finally we could test with very long expression (**performance test**), e.g. sum of 1,000,000 ones. We could generate a test of 1,000,000 numbers with the following sample code:

```
static void Main()
{
   StringBuilder expression = new StringBuilder();
   expression.Append("0");
   for (int i = 0; i < 1000000; i++)
   {
      expression.Append("+");
      expression.Append("1");
   }
   string expr = expression.ToString();
   int[] numbers = ExtractNumbers(expr);
   char[] operators = ExtractOperators(expr);
   int result = CalculateExpression(numbers, operators);
   Console.WriteLine(result);
}
```

The running time seems acceptable and the result is correct.

But what will happen if we **sum 1,000,000 times the value of 5,000,000**? We will get an **integer overflow**. We might fix this by using **long** for the sum instead of **int**:

```
private static long CalculateExpression(
   int[] numbers, char[] operators)
{
   long result = numbers[0];
   for (int i = 1; i < numbers.Length; i++)
   {
      char operation = operators[i - 1];
      int nextNumber = numbers[i];
      if (operation == '+')
      {
         result += nextNumber;
      }
      else if (operation == '-')
      {
         result -= nextNumber;
      }
   }
   return result;
}
```

After this small fix we sum 1,000,000 times the value of 5,000,000 and we **get the correct result**: 5,000,000,000,000. The problem is solved.

Exercises

1. Solve the problem "**Counting the Number of Words in a Text**" by using a buffer (**StringBuilder**) to read the text. Was there a change in the complexity of your algorithm?

2. Implement a more efficient solution to the problem "A Matrix of Prime Numbers" by using the "**Sieve of Eratosthenes**": http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes.

3. Add support for **integer multiplication and division** to the solution of "Arithmetic Expression". Bear in mind that these operations have priority over addition and multiplication!

4. Add support for **floating point numbers** in the solution of the "Arithmetic Expression" problem.

5. * Add **parentheses support** to the solution of "Arithmetic Expression". Try to process correctly the **unary minus** (e.g. in the expression -2 + 3).

6. * Write a program that validates an arithmetic expression. For example, "**2*(2.25+5.25)-17/3**" is a valid expression, but "***232*-25+(33+a**" is not.

Solutions and Guidelines

1. Your program can read from the input file **character by character**. If the current character is a **letter**, append it to the buffer, and if it is a **separator**, analyze the buffer (since it holds the current word) and clear it. When the end of the input file is reached and the file does not end with a separator, analyze the last word in the buffer. **Test the solution**!

2. First, estimate the count of prime **numbers you will need**. Then consider what the upper limit of the iterations for the **Sieve of Eratosthenes** should be, so that there are enough numbers to fill the matrix. You can come up with a **formula** by experimenting or you may use the formula from the section "Improved Performance: Sieve of Eratosthenes".

3. Taking into account that in math multiplication and division has a **priority** over addition and subtraction you can **calculate all multiplications and divisions first**, **replace** them with their result and **then all the additions and subtractions**. For example, let's the expression is "2*5-8/2+11". You may first calculate all multiplications and divisions and replace them with the results of their execution: "2*5-8/2+11" → "10-4+11". Then you may use the algorithm from the "Evaluate an Arithmetic Expression" section. Did you consider division by zero? **Test your code**. Think of special cases.

4. Floating point arithmetic can be implemented by allowing the use of the character "**.**" and replacing **int** with **double** or **decimal**. **Test your code**!

5. You can do the following: locate the first **closing parenthesis** and match it with its **corresponding opening parenthesis**. What remains between them is an arithmetic expression that can be evaluated with the same algorithm recursively. You can substitute the expression with its value and

repeat the process until there aren't any more parentheses. Eventually, you will end up with an expression without parentheses.

For example, if the expression "**2*((3+5)*(4-7*2))**" is entered, you will substitute "**(3+5)**" with **8**, and **"(4-7*2)**" with **-10**. Finally, you will replace "**(8*-10)**" with **-80** and calculate **2*-80**, thus getting the result **-160**. You will have to consider the arithmetic operations with negative numbers, i.e. adding **negative numbers support** when parsing the numbers.

There is one more algorithm. It utilizes a stack and converts the expression to **reverse Polish notation (RPN)**. Look up the terms "**postfix notation**" and **"shunting yard algorithm"** on the Internet.

To handle correctly the **unary minus**, you may consider two situations. The first is a leading unary minus (e.g. -3 + 5). The second is a minus after another operator or after a bracket, e.g. "3 * -2 + 4". The minus can be applied to a number of to an expression in brackets. In both cases you may **insert "0-" and put the number or expression on the right in brackets**. Examples:

- "-3 + 5" → "(0-3) + 5"
- "3 * -2 + 5" → "3 * (0-2) + 5"
- "-(3+2)" → "(0-(3+2))"
- "-(-1) * 3 - -1" → "(0-((0-1))) * 3 – (0-1)"

6. If you evaluate the expression using **reverse polish notation**, you can **expand your algorithm to check the expression for validity**. Follow these rules: when you expect a digit, but the next token is not a digit, then the expression is invalid; when an arithmetic operation is expected, but the next token is not a valid operator, then the expression is invalid; when the parentheses do not match, the stack will either underflow or remain non-empty at the end. Don't forget the special cases "**-1**", "**-(2+4)**", etc. **Test thoroughly your code**! There are many special cases to consider.

Chapter 26. Sample Programming Exam – Topic #3

In This Chapter

In the present chapter we will **review some sample exam problems and suggest solutions** for them. While solving the problems we will stick to the advices given in the chapter "Methodology of Problem Solving".

Problem 1: Spiral Matrix

With a given number N (input from the keyboard) generate and print a **square matrix containing the numbers from 0 to N^2-1, located as a spiral** beginning from the center of the matrix and moving clockwise starting downwards (look at the examples).

Sample output for N=3 and N=4:

4	5	6
3	0	7
2	1	8

15	4	5	6
14	3	0	7
13	2	1	8
12	11	10	9

Start Thinking on the Problem

It's obvious from the requirement that we are given an **algorithmic problem**. Contriving the appropriate algorithm for filling up the square matrix cells in the required way is the main part of the solution to the problem. We will demonstrate to the reader the typical reasoning needed for solving this particular problem.

Inventing an Idea for the Solution

The next step is to **think up the idea for the algorithm**, which we will implement. We must fill the matrix with the numbers from 0 to N^2-1 and we may immediately notice that this could be made by **a loop, which puts one of the numbers in the supposed cell of the matrix at each iteration**. We first put 0 at its place, then put 1 at its place, then put 2, and so on until we finish with putting N^2-1 at its place.

Let's **suppose we know the starting position** – the one we have to put the first number on (the zero). That's how the problem is reduced to finding a method for determining **each of the next positions**, which we must put a number at – this is **our primary subtask**.

We try to find an approach for **determining the next to the current position**: we search a strict regularity for changing the indices during the traversal of the cells. It looks like the directions of the numbers are changed from time to time, right? First the direction if down, then the direction is changed to left, later to up, then to right then again to down. Changing of the directions is **always clockwise** and the **initial direction is always downwards**.

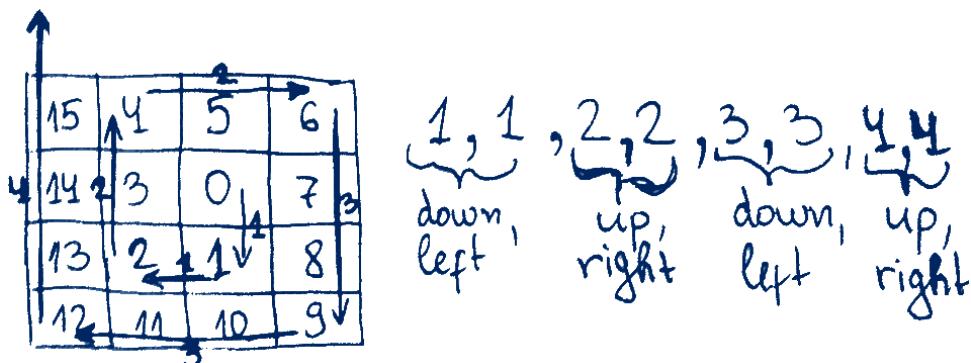
If we define an integer variable **`direction`** that holds the current moving direction, it will take sequentially the values **0** (down), **1** (left), **2** (up), **3** (right) and then again 0, 1, 2, … Looking at the problem examples (for N=3 and N=4) we can conclude that the direction stays down for some time, then changes to left, stays some time, then changes to up, stays some time, etc. We can assume that with changing the moving direction we can increase the value of **`direction`** by one and take its remainder of division by 4. Thus the next direction after 3 (right) will be 0 (down).

The next step is to determine **when the moving direction changes**: what is the number of moves in each direction. This may take some time. We can **take a sheet of paper and test few hypotheses** we might have.

From the two examples we can see that the number of moves in the consequent directions does **form special sequences**: for N=3 → 1, 1, 2, 2, 2 and for N=4 → 1, 1, 2, 2, 3, 3, 3. This means that for N=3 we move 1 cell down, then 1 cell left, then 2 cells up, then 2 cells right and finally 2 down. For N=4, the process is the same. We **found an interesting dependency** which can evolve into an algorithm for filling the spiral matrix.

If we write down a bigger matrix of the same type on a sheet of paper, we will see that the **sequence of the changes of direction** follows the same pattern: the numbers increases by 1 at an interval of two and the last number does not increase.

Seems like we have an **idea to solve the problem**: start from the middle of the matrix and move 1 cell down, 1 cell left, 2 cells up, 2 cells right, 3 cells down, 3 cells left, etc. During the moving we can fill the numbers from 0 to N^2-1 consequently at the cells we visit.

Checking the Idea

Let's **check the idea**. First we need to **find the starting cell** and check we have a correct algorithm for it. If **N is odd**, the starting cell seems to be **the absolute center cell** of the matrix. We can check this for N=1, N=3 and N=5 on a sheet of paper and this confirms to be correct. If **N is even number**, it seems like the starting cell is located **upper-right from the central point** of

the matrix. At the figure below the central point is shown for a matrix of size 4 x 4 and the starting point located at the upper-right direction:

the central point in the matrix

15	4	5	6
14	3	0	7
13	2	1	8
12	11	10	9

upper-right from the center

Now let's **check the matrix filling algorithm**. We take for example N=4. Let's start from the starting cell. The first direction is down. We go down 1 cell, then left 1 cell, then up 2 cells, then right 2 cells, then down 3 cells, then left 3 cells and finally up 3 cells. For simplicity we can assume the last step is 4 cells up but we stop at the first moment when the entire matrix if filled. The figure below shows what we could **draw on a sheet of paper** to trace how the algorithm works. See the small sketch of our algorithm, done by hand during the **idea checking** process:

After sketching the algorithm paper for N = 1, 2 and 3 **on a sheet of paper** we see that it works correctly. Seems like the idea is correct and we can thinks about how to implement it.

Data Structures and Efficiency

Let's start with choosing the **data structure** for implementing the matrix. It's appropriate to have direct access to each element of the matrix so we will choose a **two-dimensional array** **`matrix`** of integer type. When starting the program we read from the standard input the dimensionality **n** of the matrix and initialize it as it follows:

```
int[,] matrix = new int[n,n];
```

In this case the **choice of a data structure is unambiguous**. We will keep the matrix in a two-dimensional array. We have no other data. We will not have problems with the performance because the program will make as much steps as the elements in the matrix are.

Implementation of the Idea: Step by Step

We may split the implementation into few **steps**. A loop runs from 0 to N^2-1 and at each iteration it does the following steps:

- **Fill the current cell** of the matrix with the next number (this is a single move in the current direction).
- **Check whether the current direction should be changed** and if yes, change it and calculate the number of moves in the new direction.
- **Move the current position to the next cell** in the current direction (e.g. one position down / left / up / right).

Implementing the First Few Steps

We can represent the current position with integer variables **positionX** and **positionY** – the two coordinates for the position. At each iteration we will move to the next cell in the current direction and **positionX** and **positionX** will change accordingly.

For modeling the behavior of filling the spiral matrix we will use the variables **stepsCount** (total number of moves in the current direction), **stepPosition** (the move number in the current direction) and **stepChange** (flag showing if we have to change the value of **stepCount** – increments after every 2 direction changes).

Let's see how we can implement this idea as a code:

```
for (int i = 0; i < count; i++)
{
  // Fill the current cell with the current value
  matrix[positionY, positionX] = i;

  // Check for direction / step changes
  if (stepPosition < stepsCount)
  {
    stepPosition++;
  }
  else
  {
    stepPosition = 1;
    if (stepChange == 1)
    {
      stepsCount++;
    }
    stepChange = (stepChange + 1) % 2;
    direction = (direction + 1) % 4;
  }
```

```
    // Move to the next cell in the current direction
    switch (direction)
    {
        case 0:
            positionY++;
            break;
        case 1:
            positionX--;
            break;
        case 2:
            positionY--;
            break;
        case 3:
            positionX++;
            break;
    }
}
```

Performing a Partial Check after the First Few Steps

This is the moment to point out the unlikelihood of creating the body of such a loop from the first time, without making any mistakes. We already know the rule for **writing the code step by step and testing after each piece of code is written** but for the body of this loop the rule is hard to be applied – we **have no independent subproblems, which can be tested** separately of each other. To test the above code we need first to finish it: to assign initial values for all the variables used.

Assigning the Initial Values

After we have a well thought-out idea for the algorithm (even if we are not completely sure that the written code will work correctly), it remains to **set initial values** of the already defined variables and to print the matrix, obtained after the implementation of the loop.

It is clear that the number of loop iterations is exactly N^2 and that's why we replace the variable **count** with this value. From the two given examples and our own additional examples (written on a paper) we **determine the initial position** in the matrix depending on the parity (odd / even) of its size:

```
int positionX = n / 2; // The middle of the matrix
int positionY = n % 2 == 0 ? (n / 2) - 1 : (n / 2); // middle
```

To the rest of the variables we give the following initial values (we have already explained their semantics):

```
int direction = 0; // The initial direction is "down"
int stepsCount = 1; // Perform 1 step in the current direction
int stepPosition = 0; // 0 steps already performed
int stepChange = 0; // Steps count will change after 2 steps
```

Putting All Together

The **last subproblem** we have to solve for creating a working program is printing the matrix on the standard output. Let's write it, then **put all code together** and start testing.

The **fully implemented solution** is shown below. It includes reading the input data (matrix size), filling the matrix in a spiral (calculating the matrix center and filling it cell by cell) and output the result:

MatrixSpiral.cs

```
using System;

public class MatrixSpiral
{
   static void Main()
   {
      Console.Write("N = ");
      int n = int.Parse(Console.ReadLine());
      int[,] matrix = new int[n, n];

      FillMatrix(matrix, n);

      PrintMatrix(matrix, n);
   }

   private static void FillMatrix(int[,] matrix, int n)
   {
      int positionX = n / 2; // The middle of the matrix
      int positionY = n % 2 == 0 ? (n / 2) - 1 : (n / 2);

      int direction = 0; // The initial direction is "down"
      int stepsCount = 1; // Perform 1 step in current direction
      int stepPosition = 0; // 0 steps already performed
      int stepChange = 0; // Steps count changes after 2 steps

      for (int i = 0; i < n * n; i++)
      {
         // Fill the current cell with the current value
         matrix[positionY, positionX] = i;
```

```
            // Check for direction / step changes
            if (stepPosition < stepsCount)
            {
                stepPosition++;
            }
            else
            {
                stepPosition = 1;
                if (stepChange == 1)
                {
                    stepsCount++;
                }
                stepChange = (stepChange + 1) % 2;
                direction = (direction + 1) % 4;
            }

            // Move to the next cell in the current direction
            switch (direction)
            {
                case 0:
                    positionY++;
                    break;
                case 1:
                    positionX--;
                    break;
                case 2:
                    positionY--;
                    break;
                case 3:
                    positionX++;
                    break;
            }
        }
    }

    private static void PrintMatrix(int[,] matrix, int n)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                Console.Write("{0,3}", matrix[i, j]);
            }
```

```
            Console.WriteLine();
        }
    }
}
```

Testing the Solution

After we have implemented the solution it is appropriate to test it with enough values of N to ensure it works properly. We start with the **sample values 3 and 4** and then we check for 5, 6, 7, 8, 9, … It works well.

It is important to check the **border cases**: **0 and 1**. They work correctly as well. We do few more tests and we make sure all cases work correctly. We might notice that when N is large (e.g. 50) the output looks ugly, but this cannot be improved much. We can add more spaces between the numbers but the console is limited to 80 characters and the result is still ugly. We will not try to improve this further.

It is not necessary to **test the program for speed** (**performance test**, for example with N=1,000) because with a very big N the output will be extremely large and the task will be pointless.

We cannot find any non-working cases so we assume the algorithm and its implementation are both correct and the **problem is successfully solved**.

Now we are ready for the next problem from the exam.

Problem 2: Counting Words in a Text File

We are given a text file **`words.txt`**, which contains several words, one per each line. Each word consists of Latin letters only. Write a program, which **retrieves the number of matches** of each of the given words **as a substring** in the file **`text.txt`**. The counting is case insensitive. The result should be written into a text file named **`result.txt`** in the following format (the words should appear in the same order as given in the input file **`words.txt`**):

```
<word1> --> <number of matches>
<word2> --> <number of matches>
…
```

Sample input file **`words.txt`**:

```
for
academy
student
Java
develop
```

```
CAD
```

Sample input file **text.txt**:

```
The Telerik Academy for software development engineers is a
famous center for free professional training of .NET experts.
Telerik Academy offers courses designed to develop practical
computer programming skills. Students graduated the Academy are
guaranteed to have a job as a software developers in Telerik.
```

Sample result file **result.txt**:

```
for --> 2
academy --> 3
student --> 1
Java --> 0
develop --> 3
CAD --> 3
```

Below are the locations of the matched words from the above example:

```
The Telerik Academy for software development engineers is a
famous center for free professional training of .NET experts.
Telerik Academy offers courses designed to develop practical
computer programming skills. Students graduated the Academy are
guaranteed to have a job as a software developers in Telerik.
```

Start Thinking on the Problem

The emphasis of the given problem seems **not so much on the algorithm**, but on its **technical implementation**. In order to write the solution we must be familiar with working with files in C# and with the basic data structures, as well as string processing in .NET Framework.

Inventing an Idea for a Solution

We get a piece of paper, write few examples and we come up with the following **idea**: we **read the words** file, **scan through the text** and **check each word from the text** for matches with the preliminary given list of words and **increase the counter** for each matched word.

Checking the Idea

The above idea for solving the task is trivial but **we can still check it** by writing down **on a piece of paper** the sample input (words and text) and the expected result. We just scan through the text word by word in our paper

example and when we find a match with some of the preliminary given words (as a substring) we increment the counter for the matched word. The idea works in our example.

Now let's **think of counterexamples**. In the same time we might also come with few questions regarding the implementation:

- **How do we scan the text** and search for matches? We can scan the text **character by character** or **line by line** or we can **read the entire text** in the memory and then scan it in the memory (by string matching or by a regular expression). All of these approaches might work correctly but the performance could vary, right? We will think about the performance a bit later.
- **How do we extract the words** from the text? Maybe we can read the text and split it by all any non-letter characters? Where shall we take these non-letter characters from? Or we can read the text char by char and once we find a non-letter character we will have the next word from the text? The second idea seems faster and will require less memory because we don't need to read all the text at once. We should think about this, right?
- **How do we match two words?** This is a good question. Very good question. Suppose we have a word from the text and we want to match it with the words from the file **`words.txt`**. For example, we have "**Academy**" in the text and we should find whether it matches as substring the "**CAD**" word from the list of words. This will require searching each word from the list as a substring in each word from the text. Also **can we have some word appearing several times inside another?** This is possible, right?

From all the above questions we can conclude that **we don't need to read the text word by word**. We need to **match substrings**, not words! The title of the problem is misleading. It says "Counting Words in a Text File" but it should be "Counting Substrings in a Text File".

It is really good that we found we **have to match substrings** (instead of words), before we have implemented the code for the above idea, right?

Inventing a Better Idea

Now, considering the requirement for substring matching, we come with few new and probably better ideas about solving the problem:

- **Scan the text line by line** and for each line from the text and each word **check how many times the word appears as substring in the line**. The last can be counted with **`String.IndexOf(…)`** method in a loop. We already have solved this subproblem in the chapter "Strings and Text Processing" (see the section "Finding All Occurrences of a Substring").

- **Read the entire text and count the occurrences of each word in it (as a substring)**. This idea is very similar to the previous idea but it will require much memory to read the entire text. Maybe this will not be efficient. We gain nothing, but potentially we will run "out of memory".
- **Scan the text char by char** and store the read characters in a buffer. After each character read we **check if the text in the buffer ends with some of the words** from the list. We will not need to search the words in the buffer because we check for each word after each character is read. We could also clear the buffer when we read any non-letter character (because the list of words for matching should contain letters only). Thus the memory consumption will be very low.

The first and the last idea seem to be good. Which of them to implement? Maybe we could **implement both of them** and choose the faster one. Having two solutions will also improve the testing because we should get identical results with both of the solutions on all test cases.

Checking the New Ideas

We have two good ideas and we **need to check them for correctness** before thinking about implementation. How to check the ideas? We can invent a good test case on a piece of paper and try the ideas on it.

Let's have the following **list of words**:

```
Word
S
MissingWord
DS
aa
```

We might be interested to find the number of occurrences of the above words in the following **text**:

```
Word? We have few words: first word, second word, third word.
Some passwords: PASSWORD123, @PaSsWoRd!456, AAaA, !PASSWORD
```

The expected result is as follows:

```
Word --> 9
S --> 13
MissingWord --> 0
DS --> 2
aa --> 3
```

In the above example we have **many different special cases**: whole-word matching, matching as a substring, matching in different casing, matches in the start / end of the text, several matches inside the same word, overlapping

matches, etc. This example is a **very good representative of the common case** for this problem. It is important to have such **short but comprehensive test case** when solving programming problems. It is important to **have it early**, when checking the ideas, before any code is written. This avoids mistakes, catches incorrect algorithms and saves time!

Checking the Line by Line Algorithm

Now let's **check the first algorithm**: read the two lines of text and check how many times each of the words from the given list occurs in each line ignoring the character casing. At the first line we find as substrings (ignoring the case) "word" 5 times, "s" 3 times, "MissingWord" 0 times, "aa" 0 times and "ds" – 1 time. At the second line we find as substrings (ignoring the case) "word" 4 times, "s" 10 times, "MissingWord" 0 times, "aa" 3 times and "ds" – 1 time. We sum the occurrences and we find that **the result is correct**.

We **try to find counterexamples**, but we can't. The algorithm may not work with words spanning multiple lines. This is not possible by definition. It may also have issues with the overlapping matches like finding "aa" in "AAaA". This will be definitely checked after the algorithm is implemented.

Checking the Char by Char Algorithm

Let's **check the other algorithm**: scan through the text char by char, holding the characters in a buffer. After each character if the buffer ends with some of the words (ignoring the character casing), the occurrences of the matched word are increased. If a non-letter is occurred, the buffer is cleaned.

We start from **empty buffer** and **append the first char** from the text "W" to the buffer. **None of the words match the end of the buffer**. We append "o" and the buffer holds "Wo". No matches. Then we append "r". The buffer holds "Wor". Again **no matches** are found with any of the words. We **append the next** char "d" and the buffer holds "Word". We **have found a match** with the word form a list: "word". We increase the number of occurrences of the matched word from zero to one. The next char is "?" and we **clean the buffer**, because it is not a letter. The next char is " " (space). We again clean the buffer. The next char is "W". We append it to the buffer. No matches with any of the words. We continue further and further… After the last character is processed, the algorithm finishes and **the results are correct**.

We **try to find counterexamples**, but we can't. The algorithm may not work with words spanning multiple lines, but this is not possible by definition.

Decompose the Problem into Subproblems

Now let's try to divide the problem into subproblems. This should be done separately for the both algorithms we want to try because they differ significantly.

Line by Line Algorithm Decomposed into Subproblems

Let's decompose the **line by line algorithm** into subproblems (sub-steps):

1. **Read the input words**. We can **read the file `words.txt`** by using **`File.ReadAllLines(…)`**. It reads a text file in a **`string[]`** array of lines.

2. **Process the lines of the text** one by one to count the occurrences of each word in it. Initially assign zero occurrences for each word. Read the input file **`text.txt`** line by line. **For each line** from the text and **for each word check the number of its occurrences** (this is a separate subproblem) and increase the counters for each match. The occurrences counting should be case-insensitive.

3. **Count the number of occurrences of certain substring in certain text**. This is a separate subproblem. We find the leftmost occurrence of the substring in the text though **`string.IndexOf(…)`**. If the returned index > -1 (the substring exists), we increase the counter and find the next occurrence of the substring on the right from the last found index. We perform this in a loop until we find -1 as a result which means that there are no more matches. To perform case-insensitive searching we can pass a special parameter **`StringComparison.OrdinalIgnoreCase`** to the **`IndexOf()`** method.

4. **Print the results**. Process all words and for each word print it along with its counter holding its occurrences in the output file **`result.txt`**.

Char by Char Algorithm Decomposed into Subproblems

Let's decompose the **char by char algorithm** into subproblems (sub-steps):

1. **Read the input words**. We can **read the file `words.txt`** by using **`File.ReadAllLines(…)`**. It reads a text file in a **`string[]`** array of lines. The original words can be saved and a copy of them in lowercase can be made to simplify the matching with ignoring the character casing.

2. **Process the text char by char**. Read the input file **`text.txt`** and append the letters into a buffer (**`StringBuilder`**). After each letter appended check whether the text in the buffer ends with some of the words in the input list of words (this check is a separate subproblem). If so, increase the number occurrences of the matched word. If a non-letter character is found, clean the buffer. Letters are converted to lowercase before added in the buffer.

3. **Check whether a certain text (`StringBuilder`) ends by a certain string.** In case the string has length **n** lower than the length of the text, the result is **`false`**. Otherwise the **n** letters of the string should be compared one by one with the last **n** letters of the text. If a mismatch is found, the result is **`false`**. If all checks pass, the result is **`true`**.

4. **Print the results**. Process all words and for each word print it along with its counter holding its occurrences in the output file **`result.txt`**.

Think about the Data Structures

In the **line by line algorithm** we don't have any need of special data structures. We can keep the words in an **array or list of strings**. We can keep the number of occurrences for each word in **array of integer values**. The text lines we can keep in **strings**.

In the **char by char algorithm** the situation is similar. We don't need any special data structures. We can keep the words in an **array or list of strings**. We can keep the number of occurrences for each word in **array of integer values**. The buffer for the characters we can implement by **`StringBuilder`** (because we need to append chars many times).

Think about the Performance

Following the guidelines for problem solving from the chapter "Methodology of Problem Solving" we should **think about the efficiency and performance** before writing any code.

The **line by line algorithm** will process the entire text line by line and for each text line it will search for all of the words. Thus if the text has a total size of **`t`** characters and the number of words are **`w`**, the algorithm will totally perform **`w`** string searches in **`t`** characters. Each search for a word in the text will pass through the entire text (at least once, but maybe not always). If we assume that searching for a word in a text is a linear time operation, we will have **`w`** scans through the entire text, so the excepted **running time in quadratic: O(`w*t`)**. If we search in MSDN or in Internet, we will be unable to find any information about how exactly **`String.IndexOf(…)`** works internally and whether it runs in linear time or it is slower. This method calls a Win32 API function so it cannot be decompiled. Thus the best way to check its performance is by measuring.

The **char by char algorithm** will process the entire text char by char and for each character it will perform a string matching for each of the words. Suppose the text has **`t`** characters and the number of the words is **`w`**. In the average case the string matching will run in constant time (it will require just one check if the first letter is not matching, two checks if the first letter matches, etc.). In the worst case the string matching will require **`n`** comparisons where **`n`** is the length of the word being matched. Thus in the average case the expected **running time of the algorithm will be quadratic: O(`w*t`)**. In the worst case it will be **significantly slower**.

It seems like the **line by line algorithm is expected to run faster** but we are uncertain about how fast is **`string.IndexOf(…)`**, so this cannot be definitely stated. If we are at an exam, we will probably choose to implement the line by line algorithm. Just for the experiment, let's implement both of them and compare their performance.

Implementation: Step by Step

If we **directly follow the steps**, which we have already identified we can write the code with ease. Of course it is better to implement the algorithms step-by-step, to find and fix the bugs early.

Line by Line Algorithm: Step by Step Implementation

We can start implementing the line by line algorithm for word counting in a text file from the **method that counts how many times a substring appears in a text**. It should look like the following:

```
static int CountOccurrences(string substring, string text)
{
   int count = 0;
   int index = 0;
   while (true)
   {
      index = text.IndexOf(substring, index);
      if (index == -1)
      {
         // No more matches
         break;
      }
      count++;
   }
   return count;
}
```

Let’s test it before going further:

```
Console.WriteLine(
   CountOccurrences("hello", "Hello World Hello"));
```

The result is **0 – wrong**! It seems like we have **forgotten to ignore the character casing**. Let’s fix this. We need to change the name of the method as well and add the **StringComparison.OrdinalIgnoreCase** option when searching for the given substring:

```
static int CountOccurrencesIgnoreCase(
   string substring, string text)
{
   int count = 0;
   int index = 0;
   while (true)
   {
```

```
        index = text.IndexOf(substring, index,
            StringComparison.OrdinalIgnoreCase);
        if (index == -1)
        {
            // No more matches
            break;
        }
        count++;
    }
    return count;
}
```

Let's test again with the same example. **The program hangs**! What happens? We step through the code using the **debugger** and we find that the variable **index** takes the first occurrence at position **0** and at the next iteration it takes the same occurrence again at position **0** and the program enters into an endless loop. This is easy to fix. Just start searching from position **index+1** (the next position on the right), not from **index**:

```
static int CountOccurrencesIgnoreCase(
    string substring, string text)
{
    int count = 0;
    int index = 0;
    while (true)
    {
        index = text.IndexOf(substring, index + 1,
            StringComparison.OrdinalIgnoreCase);
        if (index == -1)
        {
            // No more matches
            break;
        }
        count++;
    }
    return count;
}
```

We run the fixed code with the same test. Now **the result is incorrect** (1 occurrence instead of 2). We again trace the program with the debugger and we find that the first match is at position 12. Immediately we find out why this happens: initially we start from position 1 (**index** + 1 when **index** is 0) and we skip the start of the text (position 0).

This is easy to fix:

```
static int CountOccurrencesIgnoreCase(
    string substring, string text)
{
    int count = 0;
    int index = -1;
    while (true)
    {
        index = text.IndexOf(substring, index + 1,
            StringComparison.OrdinalIgnoreCase);
        if (index == -1)
        {
            // No more matches
            break;
        }
        count++;
    }
    return count;
}
```

We test again with the same example and finally **the result is correct**. We take another, more complex test:

```
Console.WriteLine(CountOccurrencesIgnoreCase(
    "Word", "Word? We have few words: first word, second word," +
    "third word. Passwords: PASSWORD123, @PaSsWoRd, !PASSWORD"));
```

The result is **again correct** (9 matches). We test with missing word and the result is **again correct** (0 matches). This is enough. We assume the method works correctly. Now let's continue with the next step: read the words.

```
string[] words = File.ReadAllLines("words.txt");
```

There is no need to test this code. It is **too simple to have bugs**. We will test it when we test the entire solution. Let's not write the main logic of the program which **reads the text line by line and counts the occurrences of each of the input words** in each of the lines:

```
int[] occurrences = new int[words.Length];
using (StreamReader text = File.OpenText("text.txt"))
{
    string line;
    while ((line = text.ReadLine()) != null)
    {
        for (int i = 0; i < words.Length; i++)
        {
```

```
                string word = words[i];
                int wordOccurrences =
                    CountOccurrencesIgnoreCase(word, line);
                occurrences[i] += wordOccurrences;
            }
        }
}
```

This code **definitely should be tested** but it will be easier to write the code which prints the results to simplify testing. Let's do this:

```
using (StreamWriter result = File.CreateText("result.txt"))
{
    for (int i = 0; i < words.Length; i++)
    {
        result.WriteLine("{0} --> {1}", words[i], occurrences[i]);
    }
}
```

The **complete implementation** of the line by line string occurrences counting algorithms looks as follows:

CountSubstringsLineByLine.cs

```
using System;
using System.IO;

public class CountSubstringsLineByLine
{
    static void Main()
    {
        // Read the input list of words
        string[] words = File.ReadAllLines("words.txt");

        // Process the file line by line
        int[] occurrences = new int[words.Length];
        using (StreamReader text = File.OpenText("text.txt"))
        {
            string line;
            while ((line = text.ReadLine()) != null)
            {
                for (int i = 0; i < words.Length; i++)
                {
                    string word = words[i];
                    int wordOccurrences =
```

```
                CountOccurrencesIgnoreCase(word, line);
              occurrences[i] += wordOccurrences;
            }
          }
        }

        // Print the result
        using (StreamWriter result = File.CreateText("result.txt"))
        {
          for (int i = 0; i < words.Length; i++)
          {
            result.WriteLine("{0} --> {1}",
              words[i], occurrences[i]);
          }
        }
      }

      static int CountOccurrencesIgnoreCase(
        string substring, string text)
      {
        int count = 0;
        int index = -1;
        while (true)
        {
          index = text.IndexOf(substring, index + 1,
            StringComparison.OrdinalIgnoreCase);
          if (index == -1)
          {
            // No more matches
            break;
          }
          count++;
        }
        return count;
      }
    }
```

Testing the Line by Line Algorithm

Now let's **test the entire code of the program**. We try our test and **it works as expected**!

text.txt

```
Word? We have few words: first word, second word, third word.
Some passwords: PASSWORD123, @PaSsWoRd!456, AAaA, !PASSWORD
```

words.txt

```
Word
S
MissingWord
DS
aa
```

result.txt

```
Word --> 9
S --> 13
MissingWord --> 0
DS --> 2
aa --> 3
```

We also **try the sample test from the problem description** and it also **works correctly**. We try **few other tests** and all they work correctly. We try also few **border cases** like empty text and empty list of words. All these cases are handled correctly. It seems like our line by line word counting algorithm and its implementation **correctly solve the problem**.

We need to conduct only a **performance test** but let's first implement the other algorithm to be able to compare which is faster.

Char by Char Algorithm: Step by Step Implementation

Let's now implement the **char by char string occurrences counting algorithm**. We will need a **StringBuilder** to hold the characters we read and a method to check for a match at the end of the **StringBuilder**. Let's define this method first. For more flexibility it can be implemented as **extension method** to the **StringBuilder** class (recall how extension methods work from the chapter "Lambda Expressions and LINQ"):

```
static bool EndsWith(this StringBuilder buffer, string str)
{
  for (int bufIndex = buffer.Length-str.Length, strIndex = 0;
    strIndex < str.Length;
    bufIndex++, strIndex++)
  {
    if (buffer[bufIndex] != str[strIndex])
    {
      return false;
```

```
      }
   }
   return true;
}
```

Let's test the method with a sample text and its ending:

```
Console.WriteLine(
   new StringBuilder("say hello").EndsWith("hello"));
```

This test produces a **correct result: True**. Let's test the negative case:

```
Console.WriteLine(new StringBuilder("abc").EndsWith("xx"));
```

This test produces a **correct result: False**. Let's test what will happen if the ending is longer than the test:

```
Console.WriteLine(new StringBuilder("a").EndsWith("abcdef"));
```

We get **IndexOutOfRangeException**. **We found a bug**! It is easy to fix – we can return **false** if the ending string is longer than the text where it should be found:

```
static bool EndsWith(this StringBuilder buffer, string str)
{
   if (buffer.Length < str.Length)
   {
      return false;
   }
   for (int bufIndex = buffer.Length - str.Length, strIndex = 0;
      strIndex < str.Length;
      bufIndex++, strIndex++)
   {
      if (buffer[bufIndex] != str[strIndex])
      {
         return false;
      }
   }
   return true;
}
```

We **run all the tests again** and **all of them pass**. We assume the above method is correctly implemented.

Now let's continue with the step-by-step implementation. Let's implement the reading of the words:

```
string[] wordsOriginal = File.ReadAllLines("words.txt");
```

This is the same code from the line by line algorithm and it should work.

Let's now **implement the main program logic** which reads the text char by char in a buffer of characters and after each letter checks all input words for matches at the ending of the buffer:

```
int[] occurrences = new int[words.Length];
using (StreamReader text = File.OpenText("text.txt"))
{
    StringBuilder buffer = new StringBuilder();
    int nextChar;
    while ((nextChar = text.Read()) != -1)
    {
        char ch = (char)nextChar;
        if (char.IsLetter(ch))
        {
            // A letter is found --> check all words for matches
            buffer.Append(ch);
            for (int i = 0; i < words.Length; i++)
            {
                string word = words[i];
                if (buffer.EndsWith(word))
                {
                    occurrences[i]++;
                }
            }
        }
        else
        {
            // A non-letter character is found --> clean the buffer
            buffer.Clear();
        }
    }
}
```

To **test the code** we will need few lines of code to print the output:

```
using (StreamWriter result = File.CreateText("result.txt"))
{
    for (int i = 0; i < words.Length; i++)
    {
        result.WriteLine("{0} --> {1}",
            words[i], occurrences[i]);
```

```
    }
}
```

Now the program is completed and we should test it.

Testing the Char by Char Algorithm

Let's **test the entire code of the program**. We try our test and **it fails**. The produced **result is incorrect**:

```
Word --> 1
S --> 6
MissingWord --> 0
DS --> 0
aa --> 0
```

What's wrong? Maybe the **character casing**? Do we compare the characters in case-insensitive fashion? No. We found the problem.

How to fix the character casing? Maybe we need to fix the **EndsWith(…)** method. We search in MSDN and in Internet and we cannot find a method to compare case-insensitively characters. We can do something like this:

```
if (char.ToLower(ch1) != char.ToLower(ch2)) …
```

The above code will work but it will convert the characters to lowercase many times, at each character comparison. This **may be slow** so it is **better to lowercase the words and the text preliminary** before comparing. If we lowercase the words, they will be printed in lowercase at the output and this will be incorrect. So we need to remember the original words and to make a copy of them in lowercase. Let's try it. We can use the built-in extension methods from **System.Linq** to perform the lowercase conversion:

```
string[] wordsOriginal = File.ReadAllLines("words.txt");
string[] wordsLowercase =
    wordsOriginal.Select(w => w.ToLower()).ToArray();
```

We need to apply few other fixes and finally we get the following **full source code** of the char by char algorithm for counting the occurrences of a list of substrings in given text:

CountSubstringsCharByChar.cs

```
using System.IO;
using System.Linq;
using System.Text;
```

```
public static class CountSubstringsCharByChar
{
    static void Main()
    {
        // Read the input list of words
        string[] wordsOriginal = File.ReadAllLines("words.txt");
        string[] wordsLowercase =
            wordsOriginal.Select(w => w.ToLower()).ToArray();

        // Process the file char by char
        int[] occurrences = new int[wordsLowercase.Length];
        StringBuilder buffer = new StringBuilder();
        using (StreamReader text = File.OpenText("text.txt"))
        {
            int nextChar;
            while ((nextChar = text.Read()) != -1)
            {
                char ch = (char)nextChar;
                if (char.IsLetter(ch))
                {
                    // A letter is found --> check all words for matches
                    ch = char.ToLower(ch);
                    buffer.Append(ch);
                    for (int i = 0; i < wordsLowercase.Length; i++)
                    {
                        string word = wordsLowercase[i];
                        if (buffer.EndsWith(word))
                        {
                            occurrences[i]++;
                        }
                    }
                }
                else
                {
                    // A non-letter is found --> clean the buffer
                    buffer.Clear();
                }
            }
        }

        // Print the result
        using (StreamWriter result = File.CreateText("result.txt"))
        {
            for (int i = 0; i < wordsOriginal.Length; i++)
```

```
            {
                result.WriteLine("{0} --> {1}",
                    wordsOriginal[i], occurrences[i]);
            }
        }
    }

    static bool EndsWith(this StringBuilder buffer, string str)
    {
        if (buffer.Length < str.Length)
        {
            return false;
        }
        for (int bufIndex = buffer.Length-str.Length, strIndex = 0;
            strIndex < str.Length;
            bufIndex++, strIndex++)
        {
            if (buffer[bufIndex] != str[strIndex])
            {
                return false;
            }
        }
        return true;
    }
}
```

We need to **test again** with our example. Now the program works. The **result is correct**:

```
Word --> 9
S --> 13
MissingWord --> 0
DS --> 2
aa --> 3
```

We test with **all other tests** we have (the test from the problem statement, the border cases, etc.) and **all of them pass correctly**.

Testing for Performance

Now it is time to **test for performance** both our solutions. We need a **big test**. We can do it with **copy-paste**. It is easy to copy-paste the text from our text example 10,000 times and its words 100 times. The **repeating words** might cause inaccuracies in performance measuring so we manually replace the last 26 words with the letters from "a" to "z". We also play a bit with the rectangular selection in Visual Studio ([Alt] + mouse selection)

and we insert the alphabet as a vertical column in few other places. All this will result in 20,000 lines of text (1.2 MB) and 500 words (3 KB).

To **measure the execution time** we add two lines of code – before the first line of the **Main()** method and after the last line of the **Main()** method:

```
static void Main()
{
  DateTime startTime = DateTime.Now;
  // The original code goes here
  Console.WriteLine(DateTime.Now - startTime);
}
```

Now we execute first the **line by line algorithm** and it seems **not very fast**. On average computer from 2008 it prints the following result:

```
00:01:33.6393559
```

After that we **execute the char by char algorithm**. It produces the following output:

```
00:00:18.1080357
```

Unbelievable! Our char by char processing algorithm is **more than 5 times faster** than the line by line processing algorithm! But … it **still is slow**! 18 seconds for 1 MB file is not fast. How about processing 500 MB input and search for 10,000 words?

Invent a Better Idea (Again)

If we are at exam, we could **decide** whether to take the risk to **submit the char by char solution** or **spend more time to think of faster algorithm**. This depends on how much time we have to the end of the exam and how much problems we have already solved, how hard are the unsolved problems, etc. Suppose we have enough time and **we want to think more**.

What makes our solution slow? If we have 500 words, we check for each of them at each character. We do **500 * length(text)** string comparisons. The text is scanned only once (char by char). This cannot be improved, right? If we do not scan the entire text, we will be unable to find all occurrences. So if we want to improve the performance, we should look how to **check the words faster** after each character is read, right? For 500 words we perform 500 checks after each character is read. This is slow! Can't we do it faster?

In fact we perform a kind of **searching for a matching word** in a list of words? From the data structures we know that **this takes linear time**. Also, from the data structures we know that **the fastest data structure for searching is the hash-table**. OK, can't we use a hash table? Instead of

searching the words by trying each of them one by one, can't we directly find the word we need through a **hast-table lookup**?

We **take a sheet of paper** and the pencil and we start making sketches and thinking. Suppose we have the text "**passwords**" and the word "**s**". We can check the word that we obtain when we append the letters one after another:

```
p, pa, pas, pass, passw, passwo, passwor, password, passwords
```

In this case we will not match the word "**s**", right. In fact, when we find a word in the text, we should check all its substrings in the hash table. For example if the text is "**password**", **all its substrings are**:

```
p, pa, a, pas, as, s, pass, ass, ss, s, passw, assw, ssw, sw, w, passwo,
asswo, sswo, swo, wo, o, passwor, asswor, sswor, swor, wor, or, r,
password, assword, ssword, sword, word, ord, rd, d, passwords,
asswords, sswords, swords, words, ords, rds, ds, s
```

There are 45 substrings of the word "password". In a word of **n** characters we have **n*(n+1)/2** substrings. This will work well with short words (e.g. 3-4 characters) and will be slow for the long words (e.g. 15-20 characters).

We get into **another idea**? This multi-pattern matching problem should have a standard solution. Why don't we search for it in Internet? We try to **search for "multi-pattern matching algorithm" in Google** and after exploring the first few results we learn about the "Aho-Corasick string matching algorithm". Once we know the algorithm name we search for "Aho Corasick C#" and we find a nice C# implementation: https://github.com/tupunco/Tup.AhoCorasick. The theory says that after we have a new idea, we should check it for correctness. The best way to check this idea is by putting the code we found in action. In fact we do not implement the algorithm. We just try to adopt it to solve the problem we have.

Counting Substrings with the Aho-Corasick Algorithm

From the open-source implementation of the Aho-Corasick multi-pattern string matching algorithm mentioned above we can take the class **AhoCorasickSearch** and put it in action. We write a new solution of the substring counting problem based on what we have learned from the previous solutions. We find all matches of all words by the **SearchAll(…)** method of the **AhoCorasickSearch** class. Then we use a hash-table to count the number of occurrences for each of the words. To ensure we ignore the character casing we convert the text and the words into lowercase. This is the code:

CountSubstringsAhoCorasick.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.IO;

class CountSubstringsAhoCorasick
{
  static void Main()
  {
    DateTime startTime = DateTime.Now;

    // Read the input list of words
    string[] wordsOriginal = File.ReadAllLines("words.txt");
    string[] wordsLowercase =
      wordsOriginal.Select(w => w.ToLower()).ToArray();

    // Read the text
    string text = File.ReadAllText("text.txt").ToLower();

    // Find all word matches and count them
    var search = new AhoCorasickSearch();
    var matches = search.SearchAll(text, wordsLowercase);
    Dictionary<string, int> occurrences =
      new Dictionary<string, int>();
    foreach (string word in wordsLowercase)
    {
      occurrences[word] = 0;
    }
    foreach (var match in matches)
    {
      string word = match.Match;
      occurrences[word]++;
    }

    // Print the result
    using (StreamWriter result = File.CreateText("result.txt"))
    {
      foreach (string word in wordsOriginal)
      {
        result.WriteLine("{0} --> {1}", word,
          occurrences[word.ToLower()]);
      }
    }

    Console.WriteLine(DateTime.Now - startTime);
  }
```

```
}
```

We test the above code with all tests we already have and it seems to **work correctly**. We try the **performance test** and this time we can be **amazed by its speed**:

```
00:00:00.6540374
```

It **runs really fast**. This is the solution we needed and if we are allowed to use Internet at the exam, the best way to start when we have a standard well-known problem is to look for a well-known solution.

Problem 3: School

Students, which are studying in a **school**, are separated into **groups**. Each of the groups has a **teacher**. The following information is kept for the students: first name and last name. The following information is kept for the groups: name, a list of students and teacher. The following information is kept for the teachers: first name, last name and a list of groups he is teaching. Each teacher can teach more than one group. The following information is kept for the school: name, list of the teachers, list of the groups and list of the students. Your task is to:

1. **Design a set of classes** and relationships between them to model the school, its students, teachers and groups.
2. Implement **functionality for add / edit / delete** teachers, students, groups and their properties.
3. Implement **functionality for printing in human-readable form** the school, the teachers, the students, the groups and their properties.
4. Write a **sample test program**, which demonstrates the work of the implemented classes and methods.

Example of school with teachers, students and groups:

```
School "Freedom". Teachers: Tom Johnson, Elizabeth Hall.
Group "English": David Russell, Nicholas Grant, Emma Fletcher,
John Brown, Emily Cooper, teacher Elizabeth Hall.
Group "French": Kevin Simmons, Ian Hayes, teacher Elizabeth
Hall.
Group "Informatics": Jessica Carter, Andrew Cooper, Ashley
Moore, Olivia Adams, Jonathan Smith, teacher Tom Johnson.
```

Start Thinking on the Problem

This is a good example of an exam assignment the purpose of which is to test your abilities to use **object-oriented programming (OOP)** for **modeling**

problems from the real life, design classes and relationships between them as well as **working with collections**.

All we need to solve this problem is to **use our object-oriented modeling skills** that we have gained from chapter "Object-Oriented Programming Principles", especially from the section "Object-Oriented Modeling (OOM)".

Inventing an Idea for Solution

In this task there is **nothing complex to invent**. It is not algorithmic and there is not anything to be thought up. We must **define a class for each of the described in the problem description objects** (students, teachers, school students, groups, school, etc.) and after that we should define in each class **properties** to describe its characteristics and **methods** to implements the actions the class can do, e.g. printing in human-readable form. That's all.

Following the directions from the section "Object-Oriented Modeling (OOM)" we could **identify the nouns** in the problem description. Some of them should be modeled as classes; some of them as properties; and some of them may not be important and could be disregarded.

Reading the text from the problem description and analyzing the nouns, we could come to the idea to model the school through defining few interrelated classes: **`Student`**, **`Group`**, **`Teacher`** and **`School`**. For testing the classes we could create a class **`SchoolTest`**, which will create few objects of each class and will demonstrate their work in action.

Checking the Idea

We **will not check the idea** because there is nothing to be proven or checked. We need to write few classes to model a real-world situation: a school with students, teachers and groups.

Dividing the Problem into Subproblems

The implementation of each of the classes we already identified can be considered a **subproblem** of the given school modeling problem. Thus we have the following subproblems:

- Class for the students – **`Student`**. Students will have first name, last name and a method for printing in human-readable form – **`ToString()`**.

- Class for the groups – **`Group`**. Groups will have a name, a teacher and a list of students. It will also have a method for printing in human-readable form.

- Class for the teachers – **`Teacher`**. Teachers will have first name, last name and a list of groups, as well as a method for printing in human-readable form.

- Class for the school – **`School`**. It will have a name and will hold all students, all teachers and all groups.

- Class for testing the other classes – **`SchoolTest`**. It will create a school with a few students, a few groups holding subsets of the students and a few teachers. It will assign one teacher per group and a few groups per teacher accordingly. Finally the class will print the school and all its teachers, groups and students.

Think about the Data Structures

The data structures, needed for this problem, are of two main groups: **classes** and **relationships between the classes**. Classes will be classes. We have nothing to decide here. The interesting part is how to describe the relationships between the classes, e.g. when a group has a collection of students.

To **describe a relationship** (link) between two classes we can use an **array**. With an array we have access to its elements by index, but once it is created we will not be able to add or delete items (arrays have a fixed size). This makes it **uncomfortable for our problem**, because we don't know how many students we will have in the school and more students can be added or removed after the school is once created.

`List<T>` seems more comfortable. It has the advantages of an array and also has a variable length – it is easy to add or delete elements. **`List<T>`** can hold lists of students (inside the school and inside a group), lists of teachers (inside a school) and lists of groups (inside a school and inside a teacher).

So far it seems **`List<T>`** is the **most appropriate** for holding aggregations of objects inside another object. To be convinced we will analyze a few more data structures. For example **hash-table** – it is not appropriate in this case, because the school, teachers, students and groups are not of a key-value type. A hash-table would be appropriate if we need to search a student by its unique student ID, but this is not the case. Structures like **stack** and **queue** are inappropriate – we do not have LIFO or FIFO behavior.

The structure "**set**" and its implementation **`HashSet<T>`** may be used when we need to have **uniqueness** for given key. It would be good sometimes to use this structure to avoid duplicates. We must recall that **`HashSet<T>`** requires the methods **`GetHashCode()`** and **`Equals(…)`** to be correctly defined by the **`T`** type. **Shall we use sets** and where? To answer this question we need to recall the problem description. What is says? We need to design a set of classes to model the school, its students, teachers and groups and functionality for add / edit / delete teachers, students, groups and their properties. The easiest way to implement it is to hold a list of students in the school, a list of groups for each teacher, etc. **Lists are easier to implement**. Sets give uniqueness, but require **`Equals()`** and **`GetHashCode()`**. **Sets need more effort to be used**. So we may use lists to simplify our work.

According to the requirements the school should allow **add / edit / delete** of students, teachers and groups. The easiest way to implement this is to expose the lists of students, teachers and groups as public properties. **`List<T>`**

already have methods for add and delete of its elements and its elements are accessible by index and editable. It does the job.

Finally we choose to **use `List<T>` for all aggregations** in our classes and we will expose all the class members as properties with read and write access. We do not have a good reason to restrict the access to the members or implement immutable behavior.

Implementation: Step by Step

It's appropriate to start the implementation with the class **`Student`** because it does not depend on any of the other classes.

Step 1: Class Student

In the problem definition we have only two fields representing the **first name** and the **last name** of a student. We may add a property **`Name`**, which returns a string with the full name of the student and a **`ToString()`** implementation to print the student in human-readable form. We might **define the class `Student` as follows**:

Student.cs

```
public class Student
{
   public string FirstName { get; set; }
   public string LastName { get; set; }

   public Student(string firstName, string lastName)
   {
      this.FirstName = firstName;
      this.LastName = lastName;
   }

   public string Name
   {
      get
      {
         return this.FirstName + " " + this.LastName;
      }
   }

   public override string ToString()
   {
      return "Student: " + this.Name;
   }
}
```

We want to allow the class members to be editable so we define the **FirstName** and **LastName** as public read-write properties.

Testing the Class Student

Before continuing forward we want to **test the class Student** to be sure it is correct. Let's create a testing class with a **Main()** method and create a student in it and print the student:

```
class TestSchool
{
   static void Main()
   {
      Student studentPeter = new Student("Peter", "Lee");
      Console.WriteLine(studentPeter);
   }
}
```

We run the testing program and we get a **correct result**:

```
Student: Peter Lee
```

Now we can continue with the implementation of the other classes.

Step 2: Class Group

The next class we can define is **Group**. We choose it because the only one required for its definition is the class **Student**. The properties, which we will define, are the **name of the group**, a **list of the students**, which belong to the group, and a **teacher** who teaches the group. To implement the list with of the students we will use **List<Student>**. We will add a **ToString()** method to enable printing the group in a human-readable text form. Let's see the implementation of the class **Group**:

Group.cs

```
using System.Collections.Generic;

public class Group
{
   public string Name { get; set; }
   public List<Student> Students { get; set; }

   public Group(string name)
   {
      this.Name = name;
      this.Students = new List<Student>();
```

```
    }

    public override string ToString()
    {
        StringBuilder groupAsString = new StringBuilder();
        groupAsString.AppendLine("Group name: " + this.Name);
        groupAsString.Append("Students in the group: " +
            this.Students);
        return groupAsString.ToString();
    }
}
```

It is important when we create a group **to assign an empty list** of students to it. If we leave the list of students unassigned, it will be **null** and when we try to add a student, we will get an exception.

Testing the Class Group

Let's now **test the Group class**. Let's create a sample group, add few students to it and print the group at the console:

```
static void Main()
{
    Student studentPeter = new Student("Peter", "Lee");
    Student studentMaria = new Student("Maria", "Steward");
    Group groupEnglish = new Group("English language course");
    groupEnglish.Students.Add(studentPeter);
    groupEnglish.Students.Add(studentMaria);
    Console.WriteLine(groupEnglish);
}
```

We run the above testing code and **we find a bug**:

```
Group name: English language course
Students in the group:
System.Collections.Generic.List`1[Student]
```

It seems like the **list of students is printed incorrectly**. It is easy to find why. The **List<T>** class does not correctly implement **ToString()** and we need to use another way to print a list of students. We can do this with a **for**-loop but let's try something shorter and more elegant:

```
using System.Linq;
…
groupAsString.Append("Students in the group: " +
    string.Join(", ", this.Students.Select(s => s.Name)));
```

The above code uses **an extension method and a lambda expression** to select all students' names as **IEnumerable<string>** and then combines them into a string using a comma as separator. Let's **test the Group class again after the fix**:

```
Group name: English language course
Students in the group: Peter Lee, Maria Steward
```

The group class now **works correctly**.

Let's think a bit: who is teaching the students in the group? **We should have a teacher**, right. Let's try to add the simplest possible class **Teacher** and define a property of it in the **Group** class:

```
public class Teacher
{
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public string Name
  {
    get
    {
      return this.FirstName + ' ' + this.LastName;
    }
  }
}

public class Group
{
  public string Name { get; set; }
  public List<Student> Students { get; set; }
  public Teacher Teacher { get; set; }

  public Group(string name)
  {
    this.Name = name;
    this.Students = new List<Student>();
  }

  public override string ToString()
  {
    StringBuilder groupAsString = new StringBuilder();
    groupAsString.AppendLine("Group name: " + this.Name);
    groupAsString.Append("Students in the group: " +
      string.Join(", ", this.Students.Select(s => s.Name)));
```

```
        groupAsString.Append("\nGroup teacher: " +
          this.Teacher.Name);
        return groupAsString.ToString();
    }
}
```

Let's **test again** with our sample groups of two students studying English:

```
Student studentPeter = new Student("Peter", "Lee");
Student studentMaria = new Student("Maria", "Steward");
Group groupEnglish = new Group("English language course");
groupEnglish.Students.Add(studentPeter);
groupEnglish.Students.Add(studentMaria);
Console.WriteLine(groupEnglish);
```

We find **another bug**:

```
Unhandled Exception: System.NullReferenceException: Object
reference not set to an instance of an object.
   at Group.ToString() …
```

We step through the debugger and we see that we try to print the teacher's name but **there is no teacher** (it is **null**). This is easy to fix. We could check whether the teacher exists prior to printing it in the **ToString()** method:

```
if (this.Teacher != null)
{
  groupAsString.Append("\nGroup teacher: " +this.Teacher.Name);
}
```

Let's **test again after the fix**. Now we get the following correct result:

```
Group name: English language course
Students in the group: Peter Lee, Maria Steward
```

Let's now **add a teacher** to the testing group and check what happens:

```
Student studentPeter = new Student("Peter", "Lee");
Student studentMaria = new Student("Maria", "Steward");
Group groupEnglish = new Group("English language course");
groupEnglish.Students.Add(studentPeter);
groupEnglish.Students.Add(studentMaria);
Teacher teacherNatasha = new Teacher() {
  FirstName = "Natasha", LastName = "Walters" };
groupEnglish.Teacher = teacherNatasha;
```

```
Console.WriteLine(groupEnglish);
```

The **result is correct**:

```
Group name: English language course
Students in the group: Peter Lee, Maria Steward
Group teacher: Natasha Walters
```

Now the **Group** class works correctly. We can continue with the next class.

Step 3: Class Teacher

Let's define the class **Teacher**. We already have some piece of it, but let's define it in a better way. The teacher should have first name, last name and a list of group he teaches and should be printable in human-readable form. We can define it **directly repeating the logic in the Group class**:

Teacher.cs

```
public class Teacher
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public List<Group> Groups { get; set; }

  public Teacher(string firstName, string lastName)
  {
    this.FirstName = firstName;
    this.LastName = lastName;
    this.Groups = new List<Group>();
  }

  public string Name
  {
    get
    {
      return this.FirstName + " " + this.LastName;
    }
  }

  public override string ToString()
  {
    StringBuilder teacherAsString = new StringBuilder();
    teacherAsString.AppendLine("Teacher name: " + this.Name);
    teacherAsString.Append("Groups of this teacher: " +
```

```
        string.Join(", ", this.Groups.Select(s => s.Name)));
        return teacherAsString.ToString();
    }
}
```

Like in the class **Group**, it is important to create and empty list of groups instead of leaving the **Groups** property uninitialized.

Testing the Class Teacher

Before going further, let's **test the class Teacher**. We can create a teacher with a few groups and print it at the console:

```
static void Main()
{
    Teacher teacherNatasha = new Teacher("Natasha", "Walters");
    Group groupEnglish = new Group("English language");
    Group groupFrench= new Group("French language");
    teacherNatasha.Groups.Add(groupEnglish);
    teacherNatasha.Groups.Add(groupFrench);
    Console.WriteLine(teacherNatasha);
}
```

The **result is correct**:

```
Teacher name: Natasha Walters
Groups of this teacher: English language, French language
```

This was expected. We just repeated the same logic like in the **Group** class which was already tested and all bugs in it was fixed. We found once again **how important is to write the code step by step with testing and bug-fixing after each step**, right? The bug with incorrectly printing the list of students would have been repeated when printing the list of groups, right?

Step 4: Class School

We finish our object model with the **definition of the class School**, which uses all of the classes we already defined. It should have a name and should hold a list of students, a list of teachers and a list of groups:

```
public class School
{
    public string Name { get; set; }
    public List<Teacher> Teachers { get; set; }
    public List<Group> Groups { get; set; }
    public List<Student> Students { get; set; }
```

```
    public School(string name)
    {
        this.Name = name;
        this.Teachers = new List<Teacher>();
        this.Groups = new List<Group>();
        this.Students = new List<Student>();
    }
}
```

Before testing the class, let's think **what the class School is expected to do**. It should hold the students, teachers and groups and should be printable at the console, right? If we print the school, **what should be printed**? Maybe we should print its name, all its students (with their inner details), all its teachers (with their inner details) and all its groups (with their inner details). Let's try to define the **ToString()** method for the class **School**:

```
public override string ToString()
{
    StringBuilder schoolAsString = new StringBuilder();
    schoolAsString.AppendLine("School name: " + this.Name);
    schoolAsString.AppendLine("Teachers: " +
        string.Join(", ", this.Teachers.Select(s => s.Name)));
    schoolAsString.AppendLine("Students: " +
        string.Join(", ", this.Students.Select(s => s.Name)));
    schoolAsString.Append("Groups: " +
        string.Join(", ", this.Groups.Select(s => s.Name)));
    foreach (var teacher in this.Teachers)
    {
        schoolAsString.Append("\n---\n");
        schoolAsString.Append(teacher);
    }
    foreach (var group in this.Groups)
    {
        schoolAsString.Append("\n---\n");
        schoolAsString.Append(group);
    }
    foreach (var student in this.Students)
    {
        schoolAsString.Append("\n---\n");
        schoolAsString.Append(student);
    }
    return schoolAsString.ToString();
}
```

We shall **not test the class `School`**, because this will be the main purpose of our last class: **`SchoolTest`**.

Step 5: Class SchoolTest

The final thing is the implementation of the class **`SchoolTest`** the purpose of which is to demonstrate all the classes we have defined (**`Student`**, **`Group`**, **`Teacher`** and **`School`**) and their methods and properties. This is our last subproblem. For the demonstration we create a sample school with a few students, a few teachers and a few groups and we print it:

SchoolTest.cs

```
class TestSchool
{
   static void Main()
   {
      // Create a few students
      Student studentPeter = new Student("Peter", "Lee");
      Student studentGeorge = new Student("George", "Redwood");
      Student studentMaria = new Student("Maria", "Steward");
      Student studentMike = new Student("Michael", "Robinson");

      // Create a group and add a few students to it
      Group groupEnglish = new Group("English language course");
      groupEnglish.Students.Add(studentPeter);
      groupEnglish.Students.Add(studentMike);
      groupEnglish.Students.Add(studentMaria);
      groupEnglish.Students.Add(studentGeorge);

      // Create a group and add a few students to it
      Group groupJava = new Group("Java Programming course");
      groupJava.Students.Add(studentMaria);
      groupJava.Students.Add(studentPeter);

      // Create a teacher and assign it to few groups
      Teacher teacherNatasha = new Teacher("Natasha", "Walters");
      teacherNatasha.Groups.Add(groupEnglish);
      teacherNatasha.Groups.Add(groupJava);
      groupEnglish.Teacher = teacherNatasha;
      groupJava.Teacher = teacherNatasha;

      // Create another teacher and a group he teaches
      Teacher teacherSteve = new Teacher("Steve", "Porter");
      Group groupHTML = new Group("HTML course");
      groupHTML.Students.Add(studentMike);
```

```
        groupHTML.Students.Add(studentMaria);
        groupHTML.Teacher = teacherSteve;
        teacherSteve.Groups.Add(groupHTML);

        // Create a school with few students, groups and teachers
        School school = new School("Saint George High School");
        school.Students.Add(studentPeter);
        school.Students.Add(studentGeorge);
        school.Students.Add(studentMaria);
        school.Students.Add(studentMike);
        school.Groups.Add(groupEnglish);
        school.Groups.Add(groupJava);
        school.Groups.Add(groupHTML);
        school.Teachers.Add(teacherNatasha);
        school.Teachers.Add(teacherSteve);

        // Modify some of the groups, student and teachers
        groupEnglish.Name = "Advanced English";
        groupEnglish.Students.RemoveAt(0);
        studentPeter.LastName = "White";
        teacherNatasha.LastName = "Hudson";

        // Print the school
        Console.WriteLine(school);
    }
}
```

We run the program and we **get the expected result**:

```
School name: Saint George High School
Teachers: Natasha Hudson, Steve Porter
Students: Peter White, George Redwood, Maria Steward, Michael
Robinson
Groups: Advanced English, Java Programming course, HTML course
---
Teacher name: Natasha Hudson
Groups of this teacher: Advanced English, Java Programming
course
---
Teacher name: Steve Porter
Groups of this teacher: HTML course
---
Group name: Advanced English
Students in the group: Michael Robinson, Maria Steward, George
```

```
Redwood
Group teacher: Natasha Hudson
---
Group name: Java Programming course
Students in the group: Maria Steward, Peter White
Group teacher: Natasha Hudson
---
Group name: HTML course
Students in the group: Michael Robinson, Maria Steward
Group teacher: Steve Porter
---
Student: Peter White
---
Student: George Redwood
---
Student: Maria Steward
---
Student: Michael Robinson
```

Of course in real life programs do not start from the first time, but in this task the mistakes you could make are trivial so there's no point in discussing them. **All classes are implemented and tested**. We are almost finished with this problem.

Testing the Solution

As usually, it remains to **test if the entire solution is working** correctly. We've already done this. We tested all the classes in their nominal case.

We can do some tests with **the border cases**, for instance a group without students, empty school, etc. It seems like these cases **work correctly**. We might test a student without a name, but it is unclear whether the class should **keep itself of incorrect names** and what is a correct name. We can leave these classes without checks for the names. It will be a responsibility of their caller to put correct names though their constructors and properties. The problem description says nothing about this.

It is interesting how we **delete a student**. In our current implementation, if we delete a student, we will need to remove it from the school and to remove it from all groups he belongs to. The removal itself will require the student to have the **`Equals()`** method defined correctly or we should compare students by hand (property by property). It is unclear from the problem description how exactly the "delete student" operation should work.

We assume we don't have time and we submit the solution in its current state **without efficient delete operation**. Sometimes it takes too much time to fix something and it is better to leave it in not perfect form. Below is the full source code of the **solution of the school modeling problem**:

School.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class Student
{
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public Student(string firstName, string lastName)
  {
    this.FirstName = firstName;
    this.LastName = lastName;
  }

  public string Name
  {
    get
    {
      return this.FirstName + " " + this.LastName;
    }
  }

  public override string ToString()
  {
    return "Student: " + this.Name;
  }
}


public class Group
{
  public string Name { get; set; }
  public List<Student> Students { get; set; }
  public Teacher Teacher { get; set; }

  public Group(string name)
  {
    this.Name = name;
    this.Students = new List<Student>();
  }
```

```
    public override string ToString()
    {
        StringBuilder groupAsString = new StringBuilder();
        groupAsString.AppendLine("Group name: " + this.Name);
        groupAsString.Append("Students in the group: " +
            string.Join(", ", this.Students.Select(s => s.Name)));
        if (this.Teacher != null)
        {
            groupAsString.Append("\nGroup teacher: " +
                this.Teacher.Name);
        }
        return groupAsString.ToString();
    }
}

public class Teacher
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<Group> Groups { get; set; }

    public Teacher(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.Groups = new List<Group>();
    }

    public string Name
    {
        get
        {
            return this.FirstName + " " + this.LastName;
        }
    }

    public override string ToString()
    {
        StringBuilder teacherAsString = new StringBuilder();
        teacherAsString.AppendLine("Teacher name: " + this.Name);
        teacherAsString.Append("Groups of this teacher: " +
            string.Join(", ", this.Groups.Select(s => s.Name)));
        return teacherAsString.ToString();
```

```
    }
}

public class School
{
    public string Name { get; set; }
    public List<Teacher> Teachers { get; set; }
    public List<Group> Groups { get; set; }
    public List<Student> Students { get; set; }

    public School(string name)
    {
        this.Name = name;
        this.Teachers = new List<Teacher>();
        this.Groups = new List<Group>();
        this.Students = new List<Student>();
    }

    public override string ToString()
    {
        StringBuilder schoolAsString = new StringBuilder();
        schoolAsString.AppendLine("School name: " + this.Name);
        schoolAsString.AppendLine("Teachers: " +
            string.Join(", ", this.Teachers.Select(s => s.Name)));
        schoolAsString.AppendLine("Students: " +
            string.Join(", ", this.Students.Select(s => s.Name)));
        schoolAsString.Append("Groups: " +
            string.Join(", ", this.Groups.Select(s => s.Name)));
        foreach (var teacher in this.Teachers)
        {
            schoolAsString.Append("\n---\n");
            schoolAsString.Append(teacher);
        }
        foreach (var group in this.Groups)
        {
            schoolAsString.Append("\n---\n");
            schoolAsString.Append(group);
        }
        foreach (var student in this.Students)
        {
            schoolAsString.Append("\n---\n");
            schoolAsString.Append(student);
        }
        return schoolAsString.ToString();
```

```
    }
}

class TestSchool
{
    static void Main()
    {
        // Create a few students
        Student studentPeter = new Student("Peter", "Lee");
        Student studentGeorge = new Student("George", "Redwood");
        Student studentMaria = new Student("Maria", "Steward");
        Student studentMike = new Student("Michael", "Robinson");

        // Create a group and add a few students to it
        Group groupEnglish = new Group("English language course");
        groupEnglish.Students.Add(studentPeter);
        groupEnglish.Students.Add(studentMike);
        groupEnglish.Students.Add(studentMaria);
        groupEnglish.Students.Add(studentGeorge);

        // Create a group and add a few students to it
        Group groupJava = new Group("Java Programming course");
        groupJava.Students.Add(studentMaria);
        groupJava.Students.Add(studentPeter);

        // Create a teacher and assign it to few groups
        Teacher teacherNatasha = new Teacher("Natasha", "Walters");
        teacherNatasha.Groups.Add(groupEnglish);
        teacherNatasha.Groups.Add(groupJava);
        groupEnglish.Teacher = teacherNatasha;
        groupJava.Teacher = teacherNatasha;

        // Create another teacher and a group he teaches
        Teacher teacherSteve = new Teacher("Steve", "Porter");
        Group groupHTML = new Group("HTML course");
        groupHTML.Students.Add(studentMike);
        groupHTML.Students.Add(studentMaria);
        groupHTML.Teacher = teacherSteve;
        teacherSteve.Groups.Add(groupHTML);

        // Create a school with few students, groups and teachers
        School school = new School("Saint George High School");
        school.Students.Add(studentPeter);
        school.Students.Add(studentGeorge);
```

```
        school.Students.Add(studentMaria);
        school.Students.Add(studentMike);
        school.Groups.Add(groupEnglish);
        school.Groups.Add(groupJava);
        school.Groups.Add(groupHTML);
        school.Teachers.Add(teacherNatasha);
        school.Teachers.Add(teacherSteve);

        // Modify some of the groups, student and teachers
        groupEnglish.Name = "Advanced English";
        groupEnglish.Students.RemoveAt(0);
        studentPeter.LastName = "White";
        teacherNatasha.LastName = "Hudson";

        // Print the school
        Console.WriteLine(school);
    }
}
```

We **will not run performance tests** because the task is not of a computational nature which requires a fast algorithm. Operations that could be **slow** are **deleting of elements** from a collection. Creating objects, assigning their properties and adding elements to their collections of child elements are all fast operations. Only the **deletion could be slow**. We could improve its performance by using **HashSet<T>** instead of **List<T>** in all aggregations. We leave this to the reader.

Let's make just one more note. Why we did not notice **the performance problem with deleting elements** earlier? Let's recall how we proceeded with solving this problem. After thinking about the data structures we had to thing about the performance right? Did we do this step? We omitted this step and we found the problem too late. The conclusion is: follow the guidelines for problem solving. They are very wise.

Exercises

1. Write a program, which prints a **square spiral matrix** beginning from the number 1 in the upper right corner and moving clockwise. Examples for N=3 and N=4:

7	8	1
6	9	2
5	4	3

10	11	12	1
9	16	13	2
8	15	14	3
7	6	5	4

2. Write a program, which **counts the phrases in a text file**. Any sequence of characters could be given as phrase for counting, even sequences containing separators. For instance in the text "I am a student in Sofia" the phrases "s", "stu", "a" and "I am" are found respectively 2, 1, 3 and 1 times.

3. Model with OOP the **file system of a computer** running Windows. We have devices, directories and files. The **devices** are for instance floppy disk, HDD, CD-ROM, etc. They have a name and a tree of directories and files. Each **directory** has a name, date of last change and list of files and directories, which it holds. Each **file** has a name, date of creation, date of last change and content. Each file is placed in one of the directories. Each file can be text or binary. **Text files** contain text (**string**), and **the binary ones** – sequence of bytes (**byte[]**). Create a class, which tests the other classes and demonstrates how we can build a model for devices, directories and files in the computer.

4. Using the classes from the previous task write a program which **takes the real file system from your computer and loads it in your classes** (just the names of the devices, directories and files, without the content of the files because you will run out of memory).

Solutions and Guidelines

1. The task is analogical to the first task of the sample exam. You can modify the sample solution given above.

2. You may read the text **char by char** and after each char to append it to the current buffer **buf** and check each of the searched word for a match with **EndsWith()** in the buffer's end. Of course you cannot use efficiently hash-table and you will have a loop for each letter from the text, which is not the fastest solution. This is a modification of the "char by char algorithm for word counting".

   Implementing a **faster solution** needs to adapt the Aho-Corasick algorithm. Try to play with it and modify the code from the section "Counting Substrings with the Aho-Corasick Algorithm".

3. The problem is analogical with the "School" problem from the sample exam and it can be solved by using the same approach. Define classes **Device**, **Directory**, **File**, **ComputerStorage** and **ComputerStorageTest**. Think of what **properties** each of these classes has and what are the **relationships between the classes**. Create a base abstract class File and inherit it from **TextFile** and **BinaryFile**. **Test your code** with sample hierarchy of devices, files and folders. Note: a file can be listed in more than one directory at the same time (unlike in the file system).

4. Use the class **System.IO.Directory** and its static methods **GetFiles()**, **GetDirectories()** and **GetLogicalDrives()**. Traverse the files system using the **BFS** or **DFS** graph traversal algorithm. Load partially the content of **long files** (e.g. the first 128 bytes / chars) to save memory.

Conclusion

If you are reading this conclusion and if you have read carefully the entire book, then please **accept our well-deserved congratulations**! We are certain that you have earned valuable knowledge in the principles of programming that will stick for life. Even if the years pass, even if technology evolves and computers are far from their current state, the fundamental knowledge of data structures in programming and the algorithmic way of thinking as well as the experience gained in solving programming problems will always aid you, if you work in the field of information technology.

Did You Solve All Problems?

If you have **solved all problems from all chapters**, in addition to reading carefully the entire book, then you can proudly **declare yourself a programmer**. Whatever technology you pick up from now on will be child's play. Now that you have grasped the basics and fundamental principles of programming, you'll easily learn to use databases and SQL, develop Web applications and server-side software (e.g. with ASP.NET and WCF), write HTML5 applications, develop for mobile devices and whatever else you'd like. You have a **great advantage over the majority of programmers** who do not know what a hash-table is, how searching in a tree works and what algorithm complexity is. If you have really made the tremendous effort to solve all problems from the book, then you have most certainly **reached a level of fundamental understanding of the concepts of programming** and a programmer's way of thinking, which will aid you for many years.

Have You Encountered Difficulties with the Exercises?

If you haven't solved all exercise problems or at least the vast majority of them, **turn back and solve them**! Yes, it does take a lot of time, but that's the way to learn programming – with a lot of work and effort. You won't learn programming without practicing it diligently!

If you have encountered difficulties, **use the discussion forum** of the courses on fundamentals of programming at the Software Academy, which follow this book: http://forums.academy.telerik.com. Several hundred people have taken these courses and the majority of them have solved all problems and shared their solutions. So, examine them, try solving the problems and then try again without using any guides.

Many **lectures and video tutorials** have been uploaded on the book's Web site (http://www.introprogramming.info). We have **free PowerPoint slides and videos** in English and Bulgarian for each chapter of the book. They will be of great use to you, especially if this is the first time you are getting involved in programming. If you decide to teach C#, programming or data structures and algorithms, the slides and exercises will help you focus on the training and save time preparing the content. It's worth checking them out.

Also, check out the **free courses** available from Telerik Software Academy (http://academy.telerik.com). All of their lectures' study materials and video recordings have been made available for free download on each course's respective Web site. These courses are an excellent follow-up to your progress as software engineers and professionals in software development. All materials (lecture slides, exercises, demos) and some video recordings, both at this book's and at Telerik Academy's Web site, are **available in English**.

How Do You Proceed After Reading the Book?

Maybe you are wondering how you should continue your development as a software engineer. You've laid solid foundations with this book, so it won't be difficult. We can give you the following instructions:

1. **Choose a language and a programming platform**, e. g. C# + .NET Framework, Java + Java EE, Ruby + Rails or PHP + CakePHP. There's nothing wrong with giving up C#. Focus on the technologies your platform supports; you'll learn the corresponding language quickly. For example, if you choose Objective-C and iPhone / iPad / iOS / Xcode programming, the algorithmic way of thinking you have acquired with this book will help you make progress.

2. **Read a book on databases** and learn how to model your application's data using tables and relations between them. Learn how to build queries for selecting and updating data in **SQL**. Learn how to work with a database server, like **Oracle**, **SQL Server** or **MySQL**. The next natural course of action is to acquire some **ORM technology**, like ADO.NET Entity Framework, Hibernate or JPA. You might also try the **NoSQL database** systems available in the public clouds.

3. **Acquire a technology for building dynamic Web sites**. Start with a book on **HTML, CSS, JavaScript** and jQuery, or with our free course on HTML5, CSS3 and JavaScript (http://html5course.telerik.com). Then explore the web development tools your platform supports, such as ASP.NET Web Forms / **ASP.NET MVC** using the .NET Platform and C#, Servlets / JSP / JSF using the Java platform, CakePHP / Symfony / Zend Framework with PHP, Ruby on Rails using Ruby or Django using Python. Learn how to make simple Web sites with dynamic content. Try creating a Web application for mobile devices using some mobile UI toolkit.

4. **Learn to write mobile applications**. Start for example with HTML5 and **Cordova**, try to deploy your apps in the large marketplaces maintained by Google, Apple, Microsoft and Amazon. Try to learn **native mobile**

development (e.g. Java and **Android** development or Objective C and **iOS** development). Create a mobile app (e.g. some game) and deploy it in some major marketplace. Thus you will pass through the entire design / develop /publish cycle and this will give you real-world mobile development experience.

5. Take up working on a **more serious project**, like a Web market or a program for managing warehouse or accounting software. This will give you the opportunity to encounter the practical problems of practical software development. You'll gain the more valuable practical experience and you'll see for yourself that coding advanced software is much more difficult than coding simple programs.

6. **Get a job at a software company!** This is very important. If you have really solved all problems from this book, you'll easily get a job offer. By working on practical software projects you'll learn a great deal of new software technologies, unlike your colleagues, and you'll come to realize that, even though you know a lot about programming, you are only at the very beginning of your career as a software engineer. You'll only get to tackle the **challenges of team work** in practice, and acquire the tools for dealing with them by working on actual software projects at an actual work environment. You'll have to work at least for a few years until you establish yourself as a software development professional. Then, perhaps, you'll remember about this book and you'll realize that you haven't gone wrong by starting with data structures and algorithms rather than directly with Web technologies, databases and mobile development.

Free Courses at Telerik Software Academy

You can save yourself a lot of trouble and nerves, if you decide to go through all of the above steps of your development as a software engineer at **Telerik Software Academy**. You'll learn under the guidance of Svetlin Nakov and instructors with practical experience in the software industry. The Academy is the easiest and absolutely free-of-charge way to lay the foundations of your development career, but it is not the only way. Everything depends on you!

Good Luck to Everyone!

On behalf of the entire panel of authors, we wish you endless success in your career and personal life!

Svetlin Nakov,

Manager of the "Technical Training" Department, Telerik Corporation,

Telerik Software Academy – http://academy.telerik.com

August 24th, 2013

There are quite many books on C# and even more on programming. Many of them are described as the best guide or the quickest way to learn the language. This book is different, because it will show you what you need to know to achieve success. If you find the topics in this book uninteresting, probably software engineering is not for you.

Vesselin Raychev, Software Engineer at Google

I have never come across a book on programming that, simultaneously, introduces the readers to the language and hones their skills in order to solve problems. I am glad that now there is such a book, and I am sure it will be very useful to future programmers.

Vassil Bakalov, Software Engineer at Microsoft

Being a witness to the efforts put into writing this book, I am happy that the enormous energy and desire to create this new kind of book have materialized into such high quality content. The book will be useful to readers by giving them a stable foundation on which to build on; a foundation to make them enthusiastic about achieving professional development in the field of programming. This is a book that will help them make a painless and efficient start in their careers.

Vassil Terziev, Founder & Co-CEO at Telerik Corp.

Over the years, you can eventually find out, on your own, the best practices that this book recommends, but should you learn by using the trial-and-error method? This book will give you the easiest way to go in the right direction - to master the basic data structures and algorithms, to master the right way of thinking and write excellent code.

Vassil Popovski, Software Architect at VMware

This book is not just for beginners. Even programmers with several years of experience can still learn from it! I recommend it to software developers who want to find out what knowledge they might have missed. Happy reading!

Lyubomir Ivanov, Head of "Data and Mobile Applications" at Mobiltel

In the book you will find plenty of the basics of programming. A similar fundamental book in the car industry would be titled "Internal Combustion Engines".

Nichola Mihailov, Software Engineer at Microsoft

Contributors

Angel Angelov
Atanas Valchev
Blagovest Buyukliev
Boris Valkov
Boyan Dimitrov
Danail Aleksiev
Dimitar Bonev
Doncho Minkov
Doroteya Agayna
Dyanko Petkov
Franz Fischbach
George Halachev
George K. Georgiev
George S. Georgiev
Georgi Mitev
Georgi Todorov
Georgi Vaklinov
Hristo Radkov
Hristo Todorov
Ivan Nenchovski
Ivaylo Dyankov
Ivaylo Gergov
Kristian Dimitrov
Kristina Nikolova
Lachezar Bozhkov
Luchesar Cekov
Lora Borisova
Marin Georgiev
Mario Peshev
Mariyan Nenchev
Martin Gebov
Martin Radev
Martin Yankov
Momchil Rogelov
Nedjaty Mehmed
Nencho Nenchev
Nikolay Angelov
Pavel Benov
Petar Velev
Radoslav Todorov
Rumyana Topalska
Stanislav Vladimirov
Teodor Rusev
Tihomir Iliev
Todor Mitev
Vasya Stankova
Ventsi Shterev
Vesselina Raikova
Viktor Ivanov
Vladimir Amiorkov
Vladimir Tsanev
Yoan Krumov
Zhasmina Stoyanova

Web site:

http://introprogramming.info

ISBN 978-954-400-773-7